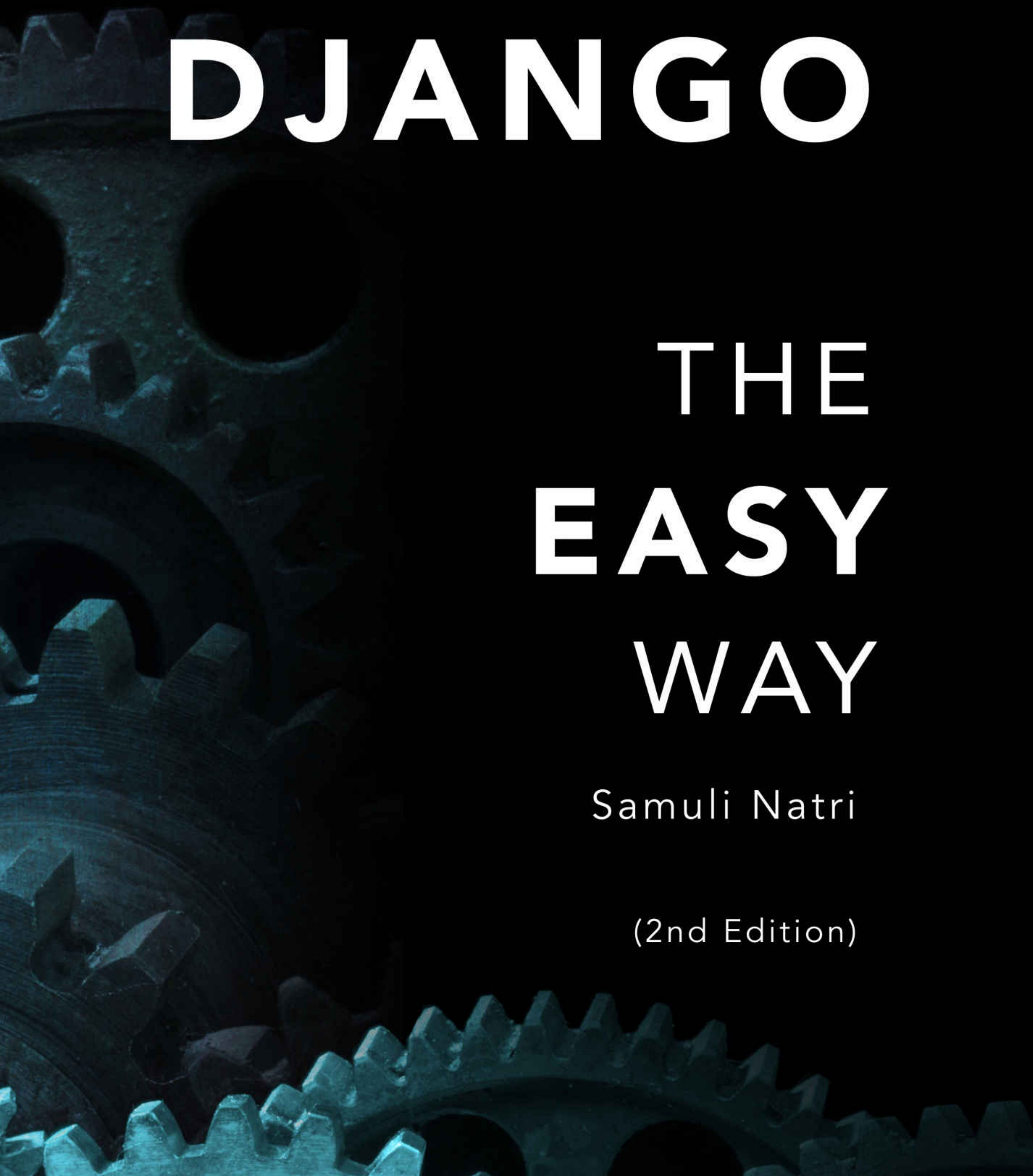


DJANGO

THE EASY WAY

Samuli Natri

(2nd Edition)

Django - The Easy Way (2nd Edition)

A step-by-step guide on building Django websites

Samuli Natri

Table of Contents

Preface

"Django - The Easy Way (2nd Edition)" book is a practical, step-by-step guide on how to build Django websites.

Django is a Python based open source **web development framework** that has been around since 2005. It enables you to create complex *database-driven* websites while keeping things decoupled and dry. The *Python Package Index* (PyPI) hosts numerous free packages that can be used to extend projects without re-inventing the wheel. Django is used by some well-known sites like Instagram, Bitbucket and Disqus.

About this book

This book is about **learning the Django web framework** with simple, practical examples. It guides you through all the main concepts one at the time. We will work on many small projects rather than working on a single big application through the book. This helps digesting the information as the projects have less distracting code from previous chapters. By the end of the book you should have a solid understanding of how to build and deploy apps with Django.

The complete book source code is available in here: https://samuli.to/Django-The-Easy-Way-Source.

Who is this book for?

This book is suitable for **beginner** to **intermediate** level web developers. You don't have to have any experience with Django or building web applications in general. We start with the very basics and increase complexity as we go along.

What this book is NOT about?

We use Bootstrap 4 to have a decent looking testing playground but otherwise *frontend* concepts are covered minimally. This is not a book about *Python*, *HTML*, *CSS* or *JavaScript*. Basic knowledge about those technologies would be helpful but is *not required* for the book. The focus is on the Django web framework **core concepts** and **deployment practices**.

How this book is organized

This book is organized in 32 chapters that focus on key concepts of the framework. I recommend reading the book in sequence, starting from the very beginning and working your way to the end from there.

Chapters 1-7

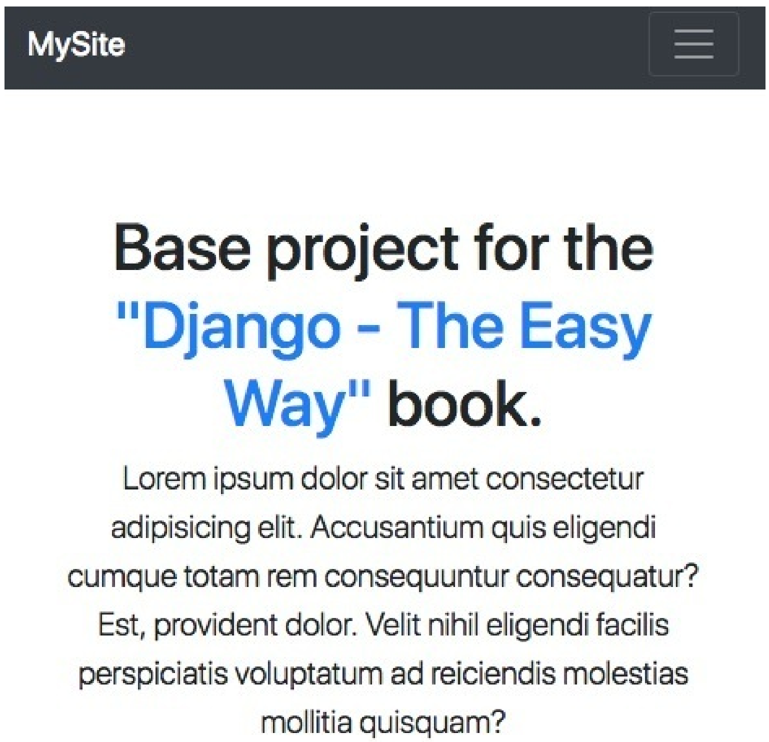
Chapters 1-7 cover how to **install Python** and use **virtual environments**.

Chapters 8-10

In chapters *8-10* we create a simple *Django project* and examine the **project structure**. “Hello world” project introduces the reader to **views**, **paths** and **templates**.

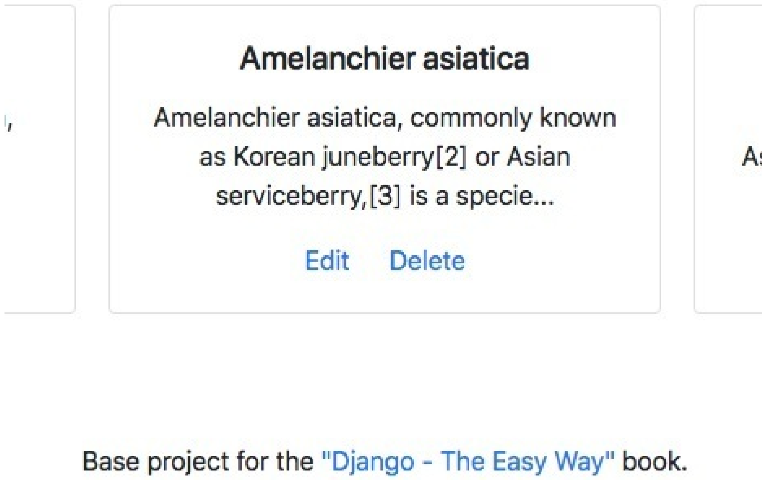
Chapters 11-13

Chapters *11-13* cover how the **template inheritance** works and how to integrate **Bootstrap 4** frontend framework with Django. We also apply custom styles with **CSS** (Cascading Style Sheets).



Chapters 14-16

Chapters *14-16* cover how to use **models** and interact with a database. We learn about **filters** and how to build a **base project** that can be used as a starting point for other projects. We create a **detail page** and learn how to work with **slugs** and **reverse URLS**.



Chapters 17-20

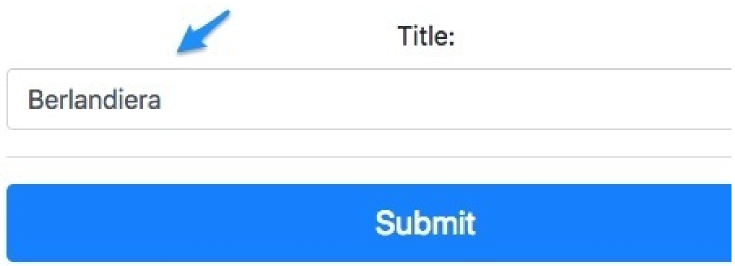
In chapters *17-20* we learn how to *categorise* items with a **ForeignKey** field and *tag* items with a **ManyToManyField**. We do **lookups** through relationships, **re-use templates** and build a minimalistic **search** feature.



Chapters 21-24

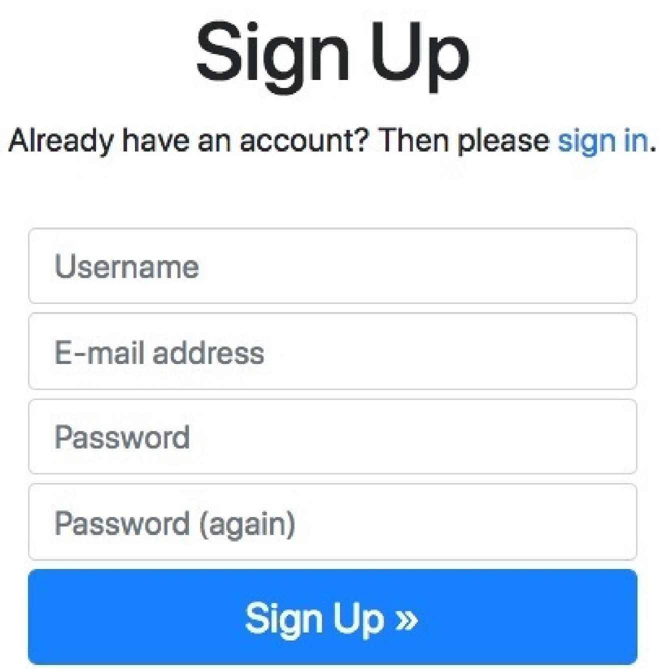
Chapters *21-24* show how to create *forms* with **ModelForm**. We **customize** the forms by changing *field order* and render *validation errors* manually. The

Python interactive interpreter is used to manipulate objects and interact with Django.



Chapters 25-26

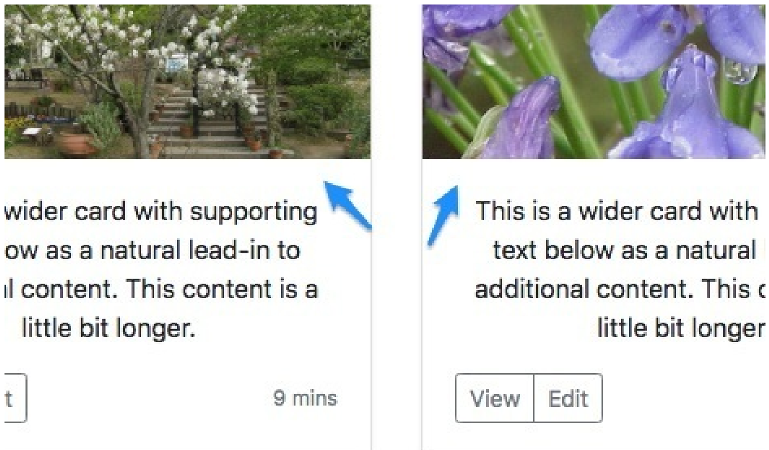
Chapters *25-26* cover how to create a complete **authentication** system with the *Allauth* package and how to theme the default forms with *Bootstrap 4*. User **authorization** is managed with *groups* and *decorators*.



Chapters 27-28

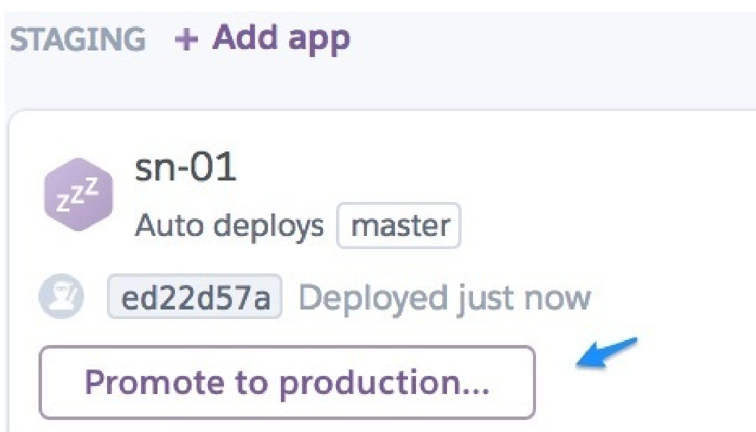
Chapters 27-28

In chapters *27-28* we **upload images** and serve them from a local media folder. Bootstrap 4 is used to create a **grid** view to display the images. The uploaded images are compressed to **thumbnails** using the *ImageKit* package.



Chapters 29-32

Chapters *29-32* show how to **deploy** to Heroku platform and serve *static* assets and *user-uploaded* files from an **Amazon AWS bucket**. We learn how to establish **continuous deployment** workflows with Heroku *pipelines* and **send emails** with *SendGrid*.



About the author

Samuli Natri has been a software developer since the 90's. He attended Helsinki University Of Technology (Computer Science) and Helsinki University (Social Sciences).

Website: https://samulinatri.com

1. Installing Python on Windows

This chapter covers

- How to *install Python* on Windows
- How to use the *interactive interpreter* to test it

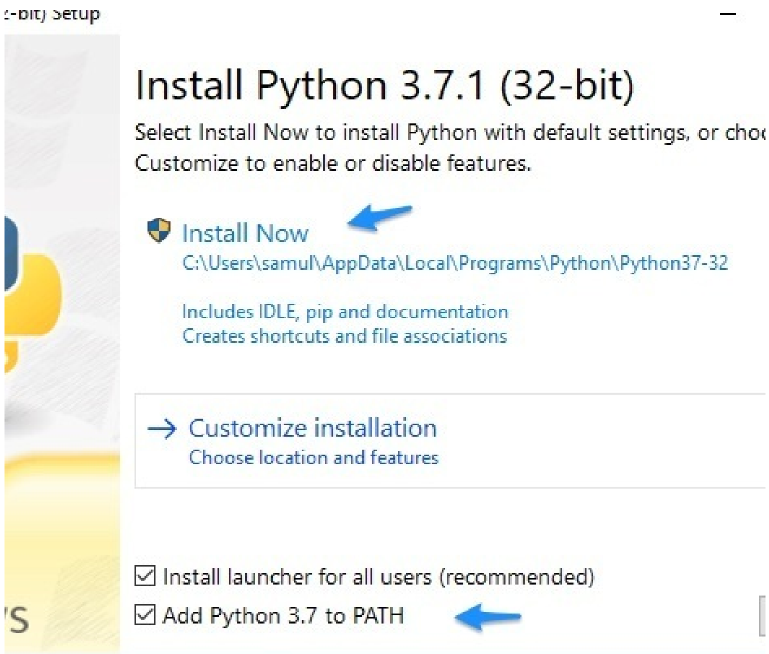
1.1 Downloading and installing Python

Visit https://samuli.to/Python-Download and download the *Windows* installer:



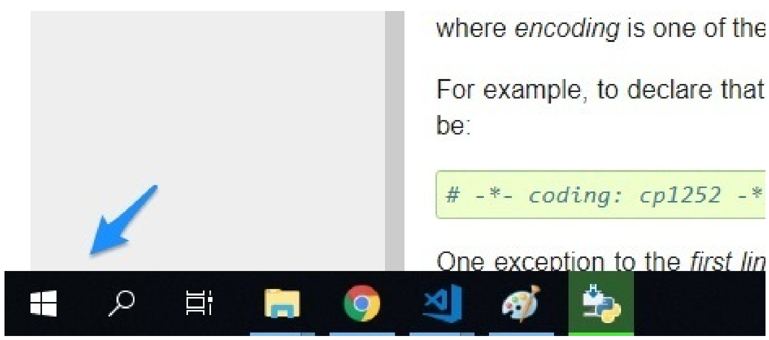
Run the installer.

Check *Add Python 3.7 to PATH* and click *Install Now*:

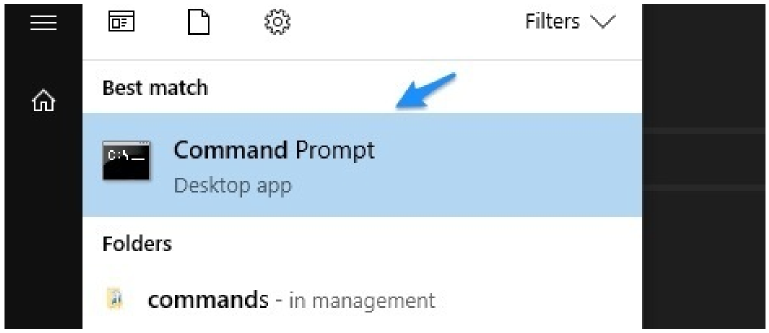


Let the installer finnish and close it.

Press *Windows* key or click the icon at the bottom left corner:



Search for *Command Prompt* and open it:



1.2 Using the interactive prompt

Type *python* in the command prompt and press enter. The *interpreter* is now in *interactive mode*, waiting for your commands:



Let's add two variables together and print out the result with `print()` function:

Interactive Python session

```
>>> a = 1
>>> b = 1
>>> c = a + b
>>> print(c)
2
>>> ^Z
```

Exit the session with *Ctrl-Z* plus *return*.

1.3 Details

1.3.1 Python interpreter

Interpreter is a software layer between your program and the computer. It reads your code and carries out the instructions it contains.

You can type and run Python code directly in the *interactive prompt*. This allows us to interact with Django projects using the command line.

1.4 Summary

- Python can easily be installed on Windows using the official *installer*.
- Make sure to add Python to the *PATH* so you can run it everywhere.
- *Interpreter* is a software layer between your code and the computer.
- You can use the *interactive prompt* to type and run Python code.

2. Installing Python on macOS

This chapter covers

- How to *install Python* on macOS
- How to use the *interactive interpreter* to test it

2.1 Downloading and installing Python

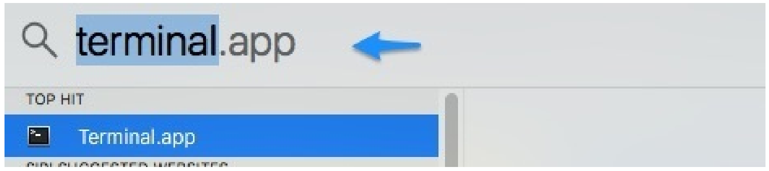
Visit https://samuli.to/Python-Download and download the latest *macOS* version:



Run the installer.

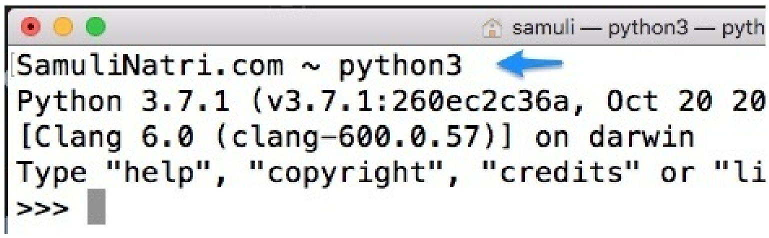


Press *Ctrl* plus *Space* and search for *terminal*:



2.2 Using the interactive prompt

Type *python3* in the terminal and press *return*. The *interpreter* is now in *interactive mode*, waiting for your commands:



Let's add two variables together and print out the result with `print()` function:

Interactive Python session

```
>>> a = 1
>>> b = 1
>>> c = a + b
>>> print(c)
2
>>> ^D
```

Exit the session with *Ctrl-D*.

2.3 Details

2.3.1 Python interpreter

Interpreter is a software layer between your program and the computer. It reads your code and carries out the instructions it contains.

You can type and run Python code directly in the *interactive prompt*. This allows us to interact with Django projects using the command line.

2.4 Summary

- Python can easily be installed on macOS using the official *installer*.
- *Interpreter* is a software layer between your code and the computer.
- You can use the *interactive prompt* to type and run Python code.

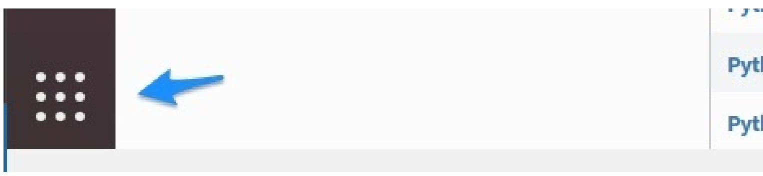
3. Installing Python on Linux

This chapter covers

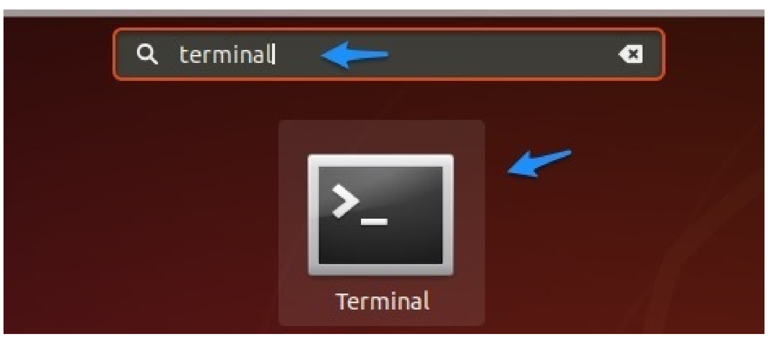
- How to *install Python* on Linux
- How to use the *interactive interpreter* to test it

3.1 Installing Python

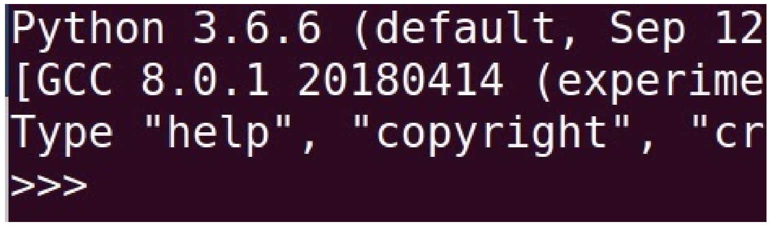
Click the *Show applications* icon at the bottom left corner:



Search for *terminal* and click the icon to open it:



Open the *Python interactive prompt* with *python3* command:



If the *python3* command doesn’t work, install it with the following command:

Terminal

```
sudo apt install python3
```

3.2 Using the interactive prompt

Type *python3* in the terminal and press enter.

The *interpreter* is now in *interactive mode*, waiting for your commands. Let’s add two variables together and print out the result with `print()` function:

Interactive Python session

```
>>> a = 1
>>> b = 1
>>> c = a + b
>>> print(c)
2
```

Exit the prompt with *Ctrl-D* plus *Enter*.

3.3 Details

3.3.1 Python interpreter

Interpreter is a software layer between your program and the computer. It reads your code and carries out the instructions it contains.

You can type and run Python code directly in the *interactive prompt*. This allows us to interact with Django projects using the command line.

3.4 Summary

- Python comes *pre-installed* on all major Linux distributions.
- *Interpreter* is a software layer between your code and the computer.

- You can use the *interactive prompt* to type and run Python code.

4. Creating virtual environments in Windows

This chapter covers

- How to create *virtual environments* in Windows

4.1 Creating and activating virtual environments

Create a new *directory* for the projects:

Terminal

```
mkdir projects
cd projects
```

venv command creates the virtual environment. Activate it with the *activate.bat* script:

Terminal

```
python -m venv venv
venv\Scripts\activate.bat
```

The *(venv)* prefix indicates that the environment is active:

Terminal

```
(venv) C:\Users\samul\projects>
```

Rest of the book will *mostly* be the same for all operating systems.

4.2 Summary

- You can use the *venv* command to create virtual environments.
- Make sure to *active* the virtual environment before you start working on a project.

5. Creating virtual environments in macOS

This chapter covers

- How to create *virtual environments* in macOS

5.1 Creating and activating virtual environments

Create a new *directory* for the projects:

Terminal

```
mkdir projects
cd projects
```

venv command creates the virtual environment. Activate it using the *source* command:

Terminal

```
python3 -m venv venv
source venv/bin/activate
```

source command reads and executes commands from a file.

The *(venv)* prefix indicates that the environment is active:

Terminal

```
(venv) ~
```

Rest of the book will *mostly* be the same for all operating systems.

5.2 Summary

- You can use the *venv* command to create virtual environments.
- Make sure to *active* the virtual environment before you start working on a project.

6. Creating virtual environments in Linux

This chapter covers

- How to create *virtual environments* in Linux

6.1 Creating and activating virtual environments

Create a new *directory* for the projects:

Terminal

```
mkdir projects
cd projects
```

venv command creates the virtual environment. Activate it using the *source* command:

Terminal

```
sudo apt-get install python3-venv
python -m venv venv
source venv\bin\activate
```

source command reads and executes commands from a file.

The *(venv)* prefix indicates that the environment is active:

Terminal

```
(venv) samuli@box:~/projects$
```

Rest of the book will *mostly* be the same for all operating systems.

6.2 Summary

- You can use the *venv* command to create virtual environments.
- Make sure to *active* the virtual environment before you start working on a project.

7. Virtual environments and pip

This chapter covers

- What are *virtual environments* and why you should use them
- How to use *pip* to manage project packages

7.1 Why use virtual environments?

Virtual environments allow you to manage project *dependencies* in an isolated manner. You can have a project that uses *Django 1.0* and another project that uses *Django 2.0*. The former project uses *Python 2* and the latter *Python 3*. With virtual environments they don't interfere which each other.

Updates may introduce changes that break your application. Maybe your favourite *package* doesn't support the new release or your own custom code is not ready for the upgrade. But at the same time you might want to start another project using the new Django release. This is where virtual environments come in handy.

Keeping all project packages in one place also makes it easier to deploy. We can generate a *requirements* list and use it to install the dependencies on another environment.

Virtual environment for each project

```
.
├── Project1
│   ├── db.sqlite3
│   ├── manage.py
│   ├── mysite
│   └── venv (With Django 1.0 + Python 2)
├── Project2
│   ├── db.sqlite3
│   ├── manage.py
│   ├── mysite
│   └── venv (With Django 2.0 + Python 3)
```

In this example each project has its own Python installation and Django package. Django is installed in the *venv* folder like any other Python package.

7.2 Details

7.2.1 Organizing folders

You don't have to put the venv folder *inside* the project folder. In fact in this book I will use one shared virtual environment for all projects. In your own real-life projects, I would recommend having a separate virtual environment for each project.

This is how we organize the projects in this book:

All projects share one virtual environment

```
└── projects
    ├── 08-Django-Project
    ├── 09-Hello-World
    ...
    └── venv
```

7.2.2 Freezing requirements

Project package list can be stored in a file using the *pip freeze* command:

Terminal

```
pip freeze > requirements.txt
```

pip is a Python *package manager*.

The *requirements.txt* file might look something like this:

requirements.txt

```
Django==2.1.3
gunicorn==19.9.0
Pillow==5.3.0
psycopg2==2.7.5
```

These dependencies can be installed using the *pip install* command:

Terminal

```
pip install -r requirements.txt
```

This installation process happens automatically when we deploy our project to the *Heroku* platform. Just make sure to *freeze* the requirements after you install or uninstall packages.

7.2.3 Excluding venv from the repository

Exclude the *venv* folder from the repository when using a *version control* system. This will be demonstrated later when we are ready to deploy.

7.2.4 Using other tools

There are other tools for managing virtual environments like *Virtualenvwrapper*. Check out this tutorial to learn more: https://samuli.to/Virtual-Environments.

7.2.5 Using python vs python3

Using a virtual environment allows us to use the *python* command (instead of *python3*) for "Python 3" regardless of the system wide Python version. If I *deactivate* the virtual environment and run *python* in macOS, it will default to *Python 2.7.10* in my machine:

Terminal

```
~ deactivate
~ python
Python 2.7.10 (default, Oct  6 2017, 22:29:07)
```

So make sure to activate the project virtual environment before you start working on it.

7.3 Summary

- *Virtual environments* allow you to manage project dependencies in an isolated manner.
- *pip* is a Python *package manager*.
- You can use the *pip freeze* command to store project dependencies list in a file.

8. Creating a Django project

This chapter covers

- How to create a *new Django project*
- How to use the built-in *development server*

8.1 Setup

Terminal

```
cd projects
mkdir 08-Django-Project
cd 08-Django-Project
source ../venv/bin/activate
```

You don't have to activate the virtual environment if it's already activated.

8.2 Creating a new Project

Install Django and use the *startproject* command to create a new Django *project*:

Terminal

```
pip install django
django-admin startproject mysite .
```

You should now have this kind of folder structure:

Project folder structure

```
projects
├── 08-Django-Project
│   ├── manage.py
│   └── mysite
└── venv
    ├── bin
    ├── include
    ├── lib
    ├── pip-selfcheck.json
    └── pyvenv.cfg
```

08-Django-Project folder is a container for the whole project. The *mysite* folder inside it is the project Python package that connects your project with Django.

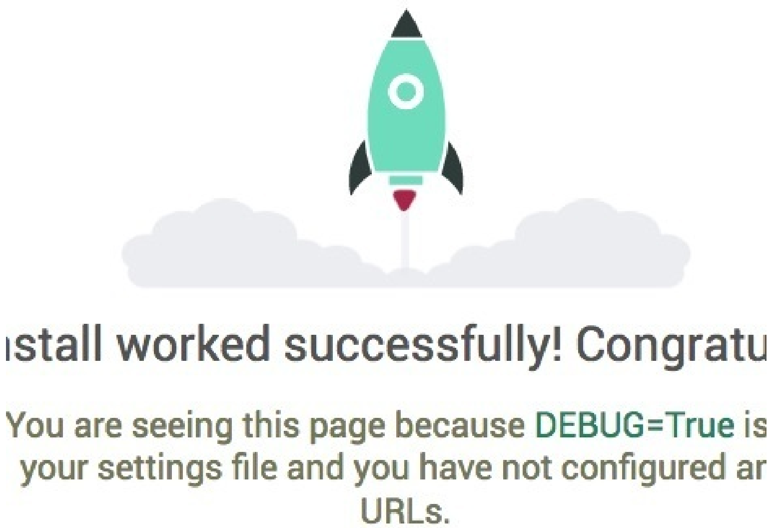
8.3 Running the development server

Use *runserver* to run the server:

Terminal

```
python manage.py runserver
```

Visit http://127.0.0.1:8000/ and you should see the welcome screen:



8.4 Details

django-admin is a command-line tool that helps you with management tasks:

Terminal

```
django-admin startproject mysite .
```

startproject command creates the Django project structure. "." denotes that we want to create the project in the *current* directory.

This also creates the *manage.py* file in the project root. *manage.py* does the same thing as *django-admin* plus it takes care of few things for you. For example, before you can use Django, you need to tell it which *settings.py* file to use. *manage.py* does this by defining an environment variable with the name "*DJANGO_SETTINGS_MODULE*". You don't have to worry about this though. Just use *manage.py* for administration tasks like this:

Terminal

```
python manage.py makemigrations
```

You might have noticed that a *database* file was generated in the project root. By default, Django is configured to use the *SQLite* database. This is perfectly fine for development purposes but for production you should consider other alternatives. With the Heroku platform we use *PostgreSQL* database.

You can ignore the “*You have 15 unapplied migration(s)..*” warning in the terminal. We will deal with migrations and databases later.

8.5 Summary

- *django-admin* is a command-line tool for administrative tasks.
- *startproject* command creates a Django project skeleton.
- It’s more convenient to use *manage.py* instead of *django-admin* for administrative tasks after the project has been created.
- *SQLite* is the default database option but you shouldn’t use it in a production environment.

9. Creating a Hello World app

This chapter covers

- How to create *apps*
- Introduction on *views*, *paths* and *templates*

9.1 Setup

Terminal

```
cp -fr 08-Django-Project 09-Hello-World
cd 09-Hello-World
source ../venv/bin/activate
```

9.2 Creating apps

Use *startapp* command to create a new app:

Terminal

```
python manage.py startapp myapp
```

Now you should have this kind of folder structure:

Folder structure

```
projects
├── 08-Django-Project
├── 09-Hello-World
│   ├── db.sqlite3
│   ├── manage.py
│   ├── myapp # < new app
│   └── mysite
└── venv
```

Edit *mysite* app *settings.py* file and add *myapp* to the *INSTALLED_APPS* list:

mysite/settings.py

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'myapp', # < here
]
```

9.3 Creating template files

Create *index.html* file in the *myapp* templates folder. You have to create the *templates* and *myapp* folders too:

Folder structure for templates

```
├── 09-Hello-World
│   ├── db.sqlite3
│   ├── manage.py
│   ├── myapp
│   │   ├── templates <-- here
│   │   │   └── myapp <-- here
│   │   │       └── index.html <-- here
```

Add this HTML markup inside the *index.html* file:

myapp/templates/myapp/index.html

```
<h1>Hello world! I was brought to you by the myapp index vi\
ew.</h1>
```

9.4 Creating views

Edit *myapp* app *views.py* file and add an *index* function:

myapp/views.py

```
from django.shortcuts import render

def index(request): # < here
    return render(request, 'myapp/index.html')
```

9.5 Adding a homepage path

Edit *mysite* app *urls.py* file add the *index* path to the *urlpatterns* list:

mysite/urls.py

```
from django.contrib import admin
from django.urls import path

from myapp import views as myapp_views # < here

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', myapp_views.index, name='index'), # < here
]
```

Run the development server:

Terminal

```
python manage.py runserver
```

Visit http://127.0.0.1:8000 and you should see this:



We will deepen the knowledge about *templates*, *views* and *paths* as we go along.

9.6 Summary

- *startapp* command creates new apps.
- Don’t forget to add the app to the *mysite/settings.py* file *INSTALLED_APPS* list.
- *app/templates/app/* is a typical location for app *template files*.
- *app/views.py* file is a typical location for app *view functions*.
- *mysite/urls.py* file is a typical location for *URL patterns*.

10. Examining the project structure and apps

This chapter covers

- What are apps?
- Overview of the project structure
- What does all the project files do?

10.1 Adding features with apps

Application (app) is a Python package that adds features to your project. With the *myapp* application we added a simple *homepage* “feature”. The project now has a custom homepage rather than the default welcome screen.

You create new apps with the *startapp* command. This creates the Django app folder structure:

Terminal

```
python manage.py startapp myapp
```

It makes sense to group similar set of features into apps. For example you could create a *forum* app that provides a forum functionality in *forum* or maybe a custom administration area in *myadmin*.

You could *potentially* re-use these apps in other projects.

The *mysite* folder that was created with the *startproject* command can also be considered an app. This app makes your Python project a *web* project.

You typically enable apps by adding a *string* to the *INSTALLED_APPS* list in the *settings.py* file:

mysite/settings.py

```
INSTALLED_APPS = [
    'django.contrib.admin',
    ...
    'myapp', # <-- here
]
```

10.2 Exploring the project structure

Let's take a closer look at an example project structure:

Project folder structure

```
├── 09-Hello-World <-- Project root
│   ├── db.sqlite3 <-- Database
│   ├── manage.py  <-- Management tool
│   ├── myapp      <-- Custom app
│   ├── forum      <-- Custom app
│   ├── myadmin    <-- Custom app
│   └── mysite     <-- Project package
└── venv <-- Virtual environment (Django + Python)
```

The project root contains the *database*, *manage.py* file and all the *apps* that are not installed in the virtual environment. *Django* package and *Python* is installed in the *venv* folder.

Here are the default contents for new apps:

Default files for a new app

```
myapp
├── __init__.py
├── __pycache__
├── admin.py
├── apps.py
├── migrations
├── models.py
├── templates
├── tests.py
└── views.py
```

`__init__.py` is usually an empty file that marks this directory as a *Python package*. Note: in newer Python versions (3.3+) it's not required to have this file: https://samuli.to/PEP-420.

`__pycache__` contains bytecode that makes the program start faster.

Django has an automatic *admin interface* in *admin* that you can use to manage content. You usually *register* models in the *admin.py* file so that they are available for management:

myapp/admin.py

```
from django.contrib import admin
from myapp.models import Post
admin.site.register(Post)
```

Don't worry about this for now. We will get back to it when we cover models. Also note that the default admin interface is intended for *internal* management purposes. You might want to allow content management with a custom solution that provides forms to add and edit content. *Custom forms* will be covered later in the book.

apps.py is used to configure the app. For example you could change the *human-readable* name for the app like this:

myapp/apps.py

```
from django.apps import AppConfig
class MyConfig(AppConfig):
    verbose_name = "Excellent App"
```

Now in the admin interface it would say "Excellent App" instead of "Myapp".

migrations folder contains the migration files for the app. These are used to apply changes to the database. You can think of the migration system as a *version control* for the database schema.

models.py file store information about the data you want to work with. Typically each model maps to a database table.

Here's an example of the *Flower* model we will use later:

myapp/models.py

```
from django.db import models

class Flower(models.Model):
    title = models.CharField(max_length=255, default='')
```

This model is mapped to a database table called *Flower* and each *attribute* like the *title* field is mapped to a database field.

Put app template files in the *templates* folder:

Templates folder

```
├── 09-Hello-World
│   ├── myapp
│   │   ├── templates
│   │   │   └── myapp
│   │   │       └── index.html # < template file
```

Templates allow you to separate the *presentation* from the application logic. Django has its own template language where you mix static HTML, variables,

tags and filters to generate the final HTML.

You typically create a subfolder for each app inside the templates folder. It might look a bit odd to have another *myapp* folder inside the templates folder but in this way we don't have to do anything special for Django to discover the template. We just have to use the right naming conventions.

For example in the myapp *views.py* file we used `myapp/index.html` as an argument for the *render* function:

myapp/views.py

```
from django.shortcuts import render

def index(request):
    return render(request, 'myapp/index.html') # here
```

With this parameter Django's template loading mechanism finds the correct template in *myapp/templates/myapp/index.html*.

tests.py is a typical place for the app testing code.

It's a convention to put *view* functions in the *views.py* file. View function takes a web *request* and returns a web *response*. In our "hello world" example the *index* view returns HTML contents generated with the help of the *index.html* template.

10.3 Exploring the project package

Let's take a look at the *project package* files:

Project package files

```
├── 09-Hello-World
│   ├── db.sqlite3
│   ├── manage.py
│   ├── myapp
│   └── mysite
│       ├── __init__.py
│       ├── __pycache__
│       ├── settings.py # < here
│       ├── urls.py # < here
│       └── wsgi.py # < here
```

Most of the project configuration happens in the *settings.py* file.

For example the default database configuration looks like this:

mysite/settings.py

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
    }
}
```

This allows you to start working with a database immediately.

For *PostgreSQL* database we would do something like this:

PostgreSQL configuration example

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql_psycopg2',
        'NAME': 'mysitedb',
        'USER': 'username',
        'PASSWORD': 'password',
        'HOST': 'localhost',
        'PORT': '',
    }
}
```

With *Heroku* platform you don't have to configure this manually though because the *django-heroku* package does it for you.

urls.py file contains URL *patterns*. Django starts going through these patterns when user requests a page and stops when a pattern matches the requested URL.

In our "Hello world!" example the *index* view will be called when user visits the homepage:

urls.py

```
from django.contrib import admin
from django.urls import path

from myapp import views as myapp_views

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', myapp_views.index, name='index'), # < here
]
```

WSGI is a specification that deals with interactions between web servers and Python web applications. The *startproject* command sets up default configuration for it in *wsgi.py*.

10.4 Summary

- *startproject* command creates a project skeleton with all the files you need to get started.
- *Project package* (folder with *settings.py* file) connects your Python project with Django.
- You typically add *features* to your project with *apps*.
- *startapp* command creates a basic application skeleton.

11. Working with template inheritance

This chapter covers

- How to setup a *base* app
- How the template *inheritance* works

11.1 Setup

Terminal

```
cp -fr 09-Hello-World 11-Template-Inheritance
cd 11-Template-Inheritance
source ../venv/bin/activate
```

11.2 Creating a base app

Create a new app:

Terminal

```
python manage.py startapp base
```

You should now have this kind of folder structure:

Folder structure

```
11-Template-Inheritance
├── base < # new app
├── db.sqlite3
├── manage.py
├── myapp
└── mysite
```

Edit *mysite* app *settings.py* file and add the *base* app to the *INSTALLED_APPS* list:

mysite/settings.py

```
INSTALLED_APPS = [
    ...
    'django.contrib.staticfiles',
    'base', # < here
    'myapp',
]
```

11.3 Extending templates

Create a *base.html* file in the base app templates folder:

Template file location

```
11-Template-Inheritance
├── base
│   ├── templates <-- here
│   │   └── base  <-- here
│   │       └── base.html <-- here
...
```

Add these lines to the *base.html* file:

base/templates/base/base.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>MySite</title>
</head>
<body>
    <div id="content">
        {% block content %}{% endblock %}
    </div>
</body>
</html>
```

Replace myapp *index.html* file contents with these lines:

myapp/templates/myapp/index.html

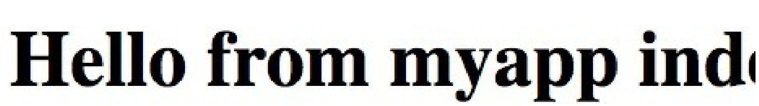
```
{% extends 'base/base.html' %}
{% block content %}
        <h1>Hello from myapp index view!</h1>
{% endblock %}
```

Run the development server:

Terminal

```
python manage.py runserver
```

Visit http://127.0.0.1:8000 to see the results:



Right-click the web page to view the *page source*:

Page source

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>MySite</title>
</head>
<body>
    <div id="content">

        <h1>Hello from myapp index view!</h1>

    </div>
</body>
</html>
```

11.4 Details

Let’s take a closer look on how this works.

Parent and child templates

```
11-Template-Inheritance
├── base
│   ├── templates
│   │   └── base
│   │       └── base.html # < parent template
├── myapp
│   ├── templates
│   │   └── myapp
│   │       └── index.html # < child template
```

With template inheritance we can have a base “skeleton” that has blocks that child templates can override.

In *base.html* we define a *content* block:

base/templates/base/base.html

```
<div id="content">
    {% block content %}{% endblock %}
</div>
```

In *index.html* we also define a *content block*:

myapp/templates/myapp/index.html

```
{% extends 'base/base.html' %}
{% block content %}
        <h1>Hello from myapp index view!</h1>
{% endblock %}
```

This block overrides the content block in the base template.

`{% extends 'base/base.html' %}` tells the templating engine that this template *extends* another template. In this case the *index.html* template extends the

base.html template.

`{% %}` marks a *tag*. These provide several kinds of features like `for` loops and inheritance related functionality.

Now we don't have to specify the common boilerplate markup for every page. This is one of the benefits you have with dynamic systems like Django.

11.5 Summary

- You can create a *base* app to hold things that are common to all apps like the main HTML skeleton.
- *Template inheritance* allows you to define blocks that child templates can override.

12. Installing Bootstrap 4 theme

This chapter covers

- How to use *Bootstrap 4* with your templates

12.1 Setup

Terminal

```
cp -fr 11-Template-Inheritance 12-Bootstrap
cd 12-Bootstrap
source ../venv/bin/activate
```

12.2 Modifying an existing template

Visit https://samuli.to/Bootstrap-Template and right-click the page to see its *source code*. Copy the source code and replace the content of the *base.html* file with it.

Replace the `<title>` element with this:

base/templates/base/base.html

```
<title>Base project for the "Django - The Easy Way" book | \
MySite</title>
```

Visit https://samuli.to/Bootstrap and copy the *BootstrapCDN* CSS link that looks like this:

Link to copy

```
<link rel="stylesheet" href="https://stackpath.bootstrapcdn\
.com/bootstrap/4.1.3/css/bootstrap.min.css" integrity="sha3\
84-MCw98/SFnGE8fJT3GXwEOngsV7Zt27NXFoaoApmYm81iuXoPkFOJwJ8E\
RdknLPMO" crossorigin="anonymous">
```

Replace these lines with the copied link:

base/templates/base/base.html

```
<!-- Bootstrap core CSS -->
<link href="../../dist/css/bootstrap.min.css" rel="styleshe\
et">
```

Replace these lines…

base/templates/base/base.html

```
<!-- Custom styles for this template -->
<link href="starter-template.css
```

…with this style element:

base/templates/base/base.html

```
<style>
    body {
        padding-top: 5rem;
    }
    .starter-template {
        padding: 3rem 1.5rem;
        text-align: center;
    }
</style>
```

In the next chapter we learn how to load static files and use a separate stylesheet file for CSS.

Change the `navbar-brand` link element to this:

base/templates/base/base.html

```
<a class="navbar-brand" href="/">MySite</a>
```

Replace the `navbar-nav mr-auto` *ul* list with this:

base/templates/base/base.html

```
<ul class="navbar-nav mr-auto">
    <li class="nav-item active">
        <a class="nav-link" href="/">Home <span class="sr-o\
nly">(current)</span></a>
    </li>
</ul>
```

Remove this search `form` element:

base/templates/base/base.html

```
<form class="form-inline my-2 my-lg-0">
...
</form>
```

Replace the `starter-template` div container contents …

base/templates/base/base.html

```
<main role="main" class="container">

    <div class="starter-template">
        <h1>Bootstrap starter ...
```

```
        <p class="lead">Use ...
    </div>

</main><!-- /.container -->
```

… with the `content` block:

base/templates/base/base.html

```
<main role="main" class="container">

    <div class="starter-template">
        {% block content %}{% endblock %} <!-- here -->
    </div>

</main><!-- /.container -->
```

Replace these three lines at the end of the *base.html* file…

base/templates/base/base.html

```
<script>window.jQuery ||  ...
<script src="../../assets ...
<script src="../../dist/j ...
```

… with the *Popper.js and jQuery* links from the Bootstrap front page:

base/templates/base/base.html

```
<script src="https://cdnjs.cloudflare.com/a ...
<script src="https://stackpath.bootstrapcdn ...
```

12.3 Updating the homepage template

Replace myapp *index.html* template contents with these lines:

myapp/templates/myapp/index.html

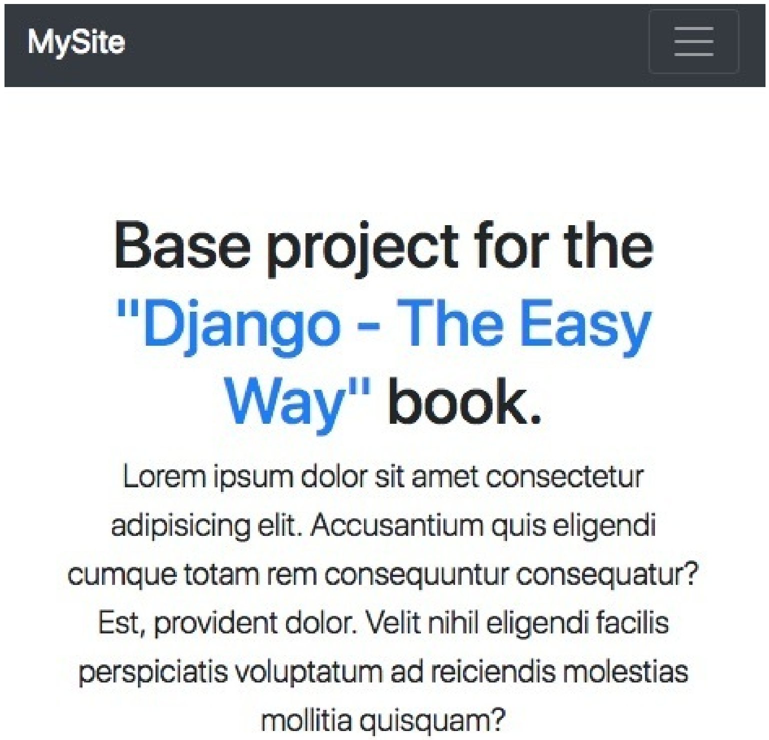
```
{% extends 'base/base.html' %}
{% block content %}

<h1>Base project for the <a target="_blank" href="https://l\
eanpub.com/django-the-easy-way">"Django - The Easy Way" </a\
>book.</h1>

<p class="lead">
    Lorem ipsum dolor sit amet consectetur adipisicing elit\
. Accusantium quis eligendi cumque totam rem consequuntur c\
onsequatur? Est, provident dolor. Velit nihil eligendi faci\
lis perspiciatis voluptatum ad reiciendis molestias molliti\
a quisquam?
</p>

{% endblock %}
```

Visit http://127.0.0.1:8000 and you should see something like this:



In this image we are seeing the mobile device styling because I resized the browser to fit everything in the image.

12.4 Details

Bootstrap is great for *prototyping* and demonstrations but it tends to result in generic looking frontends unless you modify it heavily. I personally like to build my themes from scratch with HTML, SASS and JavaScript. This book focuses on Django core concepts so I will be covering theming related topics minimally.

12.5 Summary

- It's easy to start using Bootstrap 4 with Django by modifying an existing theme.

13. Managing static files

This chapter covers

- How to add a CSS *stylesheet* file
- How to use the *static* template tag
- How to force CSS *cache* refresh

13.1 Setup

Terminal

```
cp -fr 12-Bootstrap 13-Static-Files-CSS
cd 13-Static-Files-CSS
source ../venv/bin/activate
```

13.2 Creating a stylesheet file

Create a *staticbase/css/site.css* file in the base app folder. You have to create the folder structure manually:

Stylesheet file location

```
base
├── static # < here
│   └── base # < here
│       └── css # < here
│           ├── site.css # < here
```

Edit *base.html* file and copy the contents of the *style* element to the *site.css* file. Let's also add a bright red color for `h1` elements so we can see that the CSS is working. The *site.css* file should now look like this:

base*static*base/css/site.css

```
body {
    padding-top: 5rem;
}
.starter-template {
    padding: 3rem 1.5rem;
    text-align: center;
}
h1 {
    color: red;
}
```

Replace the *style* element in the *base.html* template…

base/templates/base/base.html

```
<style>
...
</style>
```

…with this line:

base/templates/base/base.html

```
<link rel="stylesheet" href="{% static 'base/css/site.css' \
%}">
```

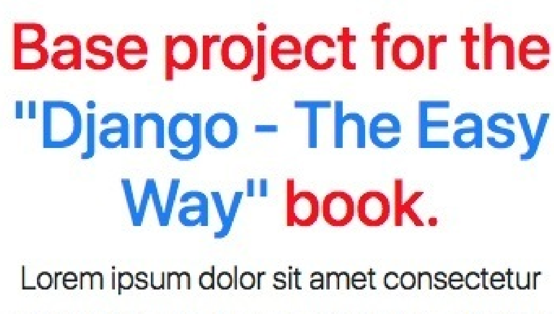
Make sure to put this *link* element *after* the line that loads the `bootstrap.min.css` file.

Make the `static` tag available in the template by using the `load` tag on top of the *base.html* file:

base/templates/base/base.html

```
{% load static %} <!-- here -->
<!doctype html>
<html lang="en">
```

`h1` elements should now be red:



You can now remove the red styling from the *site.css* file.

13.3 Details

13.3.1 Working with static files

Files like *CSS*, *JavaScript* and *images* are referred as *static files*. With *images* I mean static assets like *background* images, not *user-uploaded* files. We will deal with *media* files later when we allow users to upload files.

The *django.contrib.staticfiles* app helps you manage these static assets. It's installed by default:

mysite/settings.py

```
INSTALLED_APPS = [
        ...
    'django.contrib.staticfiles', # < here
    'base',
    'myapp',
]
```

With the development server the static files will be served automatically in *debug* mode. In production we will use the *collectstatic* command to collect all static files in one place. They are then typically served with something like *Nginx* from a single location like *static*:

Media and static files in production environment

```
├── media
│   └── images
│       ├── Agapanthus_africanus1.jpg
│       ...
├── mysite
│   ├── base
│   ├── db.sqlite3
│   ├── manage.py
├── static # < here
```

Later I will also show you how to serve these files from an *Amazon AWS* bucket.

13.3.2 Using the static tag

`load` tag loads tags and filters registered in other libraries. In this case we use it to enable the `static` tag for the template. You have to use `{% load static %}` in every template that uses the `static` tag. Even if the *parent* template already loads it.

`static` tag generates absolute URLS for the static files.

This…

Using static tag in templates

```
href="{% static 'base/css/site.css' %}"
```

…becomes this:

The resulting HTML

```
href="staticbase/css/site.css"
```

This might seem unnecessary because we could just *hard-code* the correct URL there: `staticbase/css/site.css`. But we *could* also be serving the static files from some other URL. With a proper configuration the same `static` tag could be generating these kind of links:

Serving static files from external location

```
https://static.mysite.com/base/css/site.css

OR

https://mysite.s3.amazonaws.comstaticbase/css/site.css
```

Changing this URL will be trivial since we are not hard-coding it in template files.

In general you should *avoid hard-coding* in templates when Django can generate the markup for you. This is especially helpful when providing URLS to *views* and *translating* paths.

13.3.3 Forcing cache refresh with versioning

You can also visit the style URL directly to see if the style file is served correctly:

Visiting the stylesheet path directly

```
/static/base/css/site.css
```

If you are not seeing styling changes even if the *site.css* seems to be working, your browser might be serving you *stale* content from a *cache*. In Chrome you can do this:

- Visit *View* > *Developer* > *Developer Tools*.
- Select *Network* and *Disable cache*.
- Keep the Developer Tools open.

There are similar *Developer* tools in all major browsers.

You can also *force* CSS refresh by adding a new GET parameter `?v=2` each time you make styling changes:

CSS versioning

```
<link rel="stylesheet" href="{% static 'base/css/site.css' \
%}?v=2">
```

Better yet is to let Django generate a *hash* with *ManifestStaticFileStorage*: https://samuli.to/CSS-Versioning.

13.4 Summary

- You can override Bootstrap theming with *custom stylesheets*.
- `static` tag generates absolute URLS for static assets like CSS and JavaScript files.
- In local development it’s useful to disable browser *caching*.
- In production environment it’s a common technic to add a *hash* to the CSS link path so the stylesheet is not loaded from the visitor’s browser cache.
- Static files can also be served from an external location like *Amazon AWS* bucket.

14. Creating models

This chapter covers

- How to create and use models
- How to make database queries

14.1 Setup

Terminal

```
cp -fr 13-Static-Files-CSS 14-Models
cd 14-Models
source ../venv/bin/activate
```

14.2 Creating the Flower model

Edit myapp *models.py* file:

myapp/models.py

```
├── 14-Models
│   ├── myapp
│   │   ├── models.py # < here
```

Add a *Flower* class and a *title* attribute:

myapp/models.py

```
from django.db import models

class Flower(models.Model):
    title = models.CharField(max_length=255, default='')
```

Edit myapp *admin.py* file and *register* the *Flower* class:

myapp/admin.py

```
from django.contrib import admin

from myapp.models import Flower

admin.site.register(Flower)
```

Apply changes to the database and create a *superuser*:

Terminal

```
python manage.py makemigrations
python manage.py migrate
python manage.py runserver
python manage.py createsuperuser
```

You can use *admin* as the username and password. Just bypass the validation:

Terminal

```
Bypass password validation and create user anyway? [y/N]: y
Superuser created successfully.
```

Visit http://127.0.0.1:8000*admin* and add a few flowers. Here are some examples from Wikipedia:

- https://samuli.to/Amelanchier-alnifolia
- https://samuli.to/Amelanchier-asiatica
- https://samuli.to/Agapanthus



"Flower object (n)" is not very descriptive representation for a Flower object. Let's show the *title* instead.

Edit *models.py* file and add a `__str__` method:

myapp/models.py

```
from django.db import models

class Flower(models.Model):
    title = models.CharField(max_length=255, default='')

    def __str__(self):
        return self.title
```

Now we can see the actual titles:



14.3 Listing flowers

Let’s list the flowers on the frontpage. Edit myapp *index.html* template and replace the contents with these lines:

myapp/templates/myapp/index.html

```
{% extends 'base/base.html' %}
{% block content %}

{% for flower in flowers %}
  <div class="card">
    <div class="card-body">
      <h5 class="card-title">{{ flower.title }}</h5>
      <p class="card-text">Lorem ipsum, dolor sit amet cons\
ectetur adipisicing elit.</p>
      <a href="adminmyapp/flower/{{ flower.id }}/change/"\
 class="card-link">Edit</a>
      <a href="adminmyapp/flower/{{ flower.id }}/delete/"\
 class="card-link">Delete</a>
    </div>
  </div>
{% endfor %}

{% endblock %}
```

Edit the myapp *views.py* file and replace the contents with these lines:

myapp/views.py

```
from django.shortcuts import render

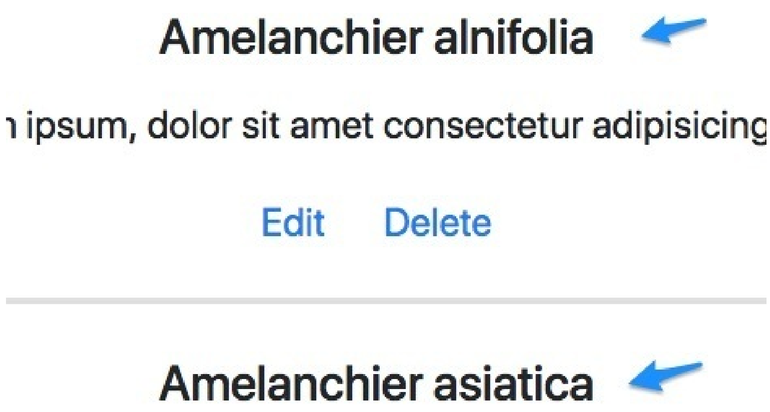
from myapp.models import Flower

def index(request):

    flowers = Flower.objects.all()

    return render(request, 'myapp/index.html', {'flowers': \
flowers })
```

Now the frontpage looks something like this:



For now the *edit* and *delete* functionality is provided through the *admin* user interface.

14.4 Details

14.4.1 Explaining models

Models offer an abstracted way to interact with data. With Django's *database-access API* you can use `Flower.objects.all()` to get all Flowers rather than doing queries like `"SELECT * FROM Flowers"`.

To create models we subclass *django.db.models.Model*:

myapp/models.py

```
from django.db import models

class Flower(models.Model):
    title = models.CharField(max_length=255, default='')
```

We *import* other modules to get access to the code they contain.

- *Flower* class represents a database table.
- *title* attribute represents a database field.

CharField is used for smaller size strings. Use *TextField* for larger texts.

To make a model editable in the admin interface, you have to *register* it as we did in the myapp *admin.py* file:

myapp/admin.py

```
admin.site.register(Flower)
```

Makemigrations command creates the migration files. These files are usually moved with rest of the code and applied in other environments:

Terminal

```
python manage.py makemigrations
```

`migrate` command updates the database schema. This will create the *Flower* table and *title* field:

Terminal

```
python manage.py migrate
```

createsuperuser command creates the main administration account. This user has all permissions by default. Make sure to use a *decent password* and *unique username* in the production server:

Terminal

```
python manage.py createsuperuser
```

14.4.2 Returning a string representation

`__str__` method returns a *human-readable representation* of an object. In this case we use the title attribute to create it:

myapp/models.py

```
def __str__(self):
    return self.title
```

You could also *format* the return string using multiple fields like this:

Formatting the representation

```
def __str__(self):
    return f"Title: {self.title}, Date: {self.date}"
```

14.4.3 Making database queries

Now that we have models, we can interact with the database using an API. `Flower.objects.all()` returns a *QuerySet* that contains all Flower objects in

the database:

Fetch objects from a database

```
flowers = Flower.objects.all()
```

In the myapp *views.py* file we pass the flowers *QuerySet* to the template using `{'flowers': flowers }`:

myapp/views.py

```
def index(request):

    flowers = Flower.objects.all()

    return render(request, 'myapp/index.html', {'flowers': \
flowers })
```

In the template we use a *for* loop to go through all the objects:

myapp/templates/myapp/index.py

```
{% for flower in flowers %}
    {{ flower.title }}
{% endfor %}
```

14.5 Summary

- Django's *database-access API* makes it easy to interact with persistent data.
- You have to register a model with `admin.site.register()` to make it available in the admin interface.
- `__str__` is used to compute a human-readable representation of an object. You can see it in use in the admin interface.
- You can use a `for` loop to iterate through a *QuerySet* in templates.

15. Creating a base project

This chapter covers

- How to prepare a general *base project*

15.1 Setup

Terminal

```
cp -fr 14-Models 15-Base-Project
cd 15-Models
source ../venv/bin/activate
```

15.2 Adding a description field

Open myapp *models.py* file:

myapp/models.py

```
├── 15-Base-Project
│   ├── myapp
│   │   ├── models.py # < here
```

Add the *description* field:

myapp/models.py

```
from django.db import models

class Flower(models.Model):
    title = models.CharField(max_length=255, default='')
    description = models.TextField(default='') # < here
```

Run migrations:

Terminal

```
python manage.py makemigrations
python manage.py migrate
```

Visit http://127.0.0.1:8000/admin/ and add descriptions for the flowers. You can find mock data in here: https://samuli.to/Lorem.

15.3 Adding masonry like columns

Edit myapp *index.html* template and wrap the cards in `card-columns` div and use the description attribute for the card text:

myapp/templates/myapp/index.html

```
<div class="card-columns"> <!-- here -->
{% for flower in flowers %}
  <div class="card">
    <div class="card-body">
      <h5 class="card-title">{{ flower.title }}</h5>
      <p class="card-text">{{ flower.description | truncate\
chars:100 }}</p> <!-- here -->
          ...
    </div>
  </div>
{% endfor %}
</div>
```

`card-columns` organizes the cards in a *masonry* like columns.

`truncatechars` *filter* truncates a string if it's longer than the number specified. It also adds an *ellipsis sequence* to the end.

15.4 Adding a footer

Add *footer* element to the *base.html* template:

base/templates/base/base.html

```
...
</main>

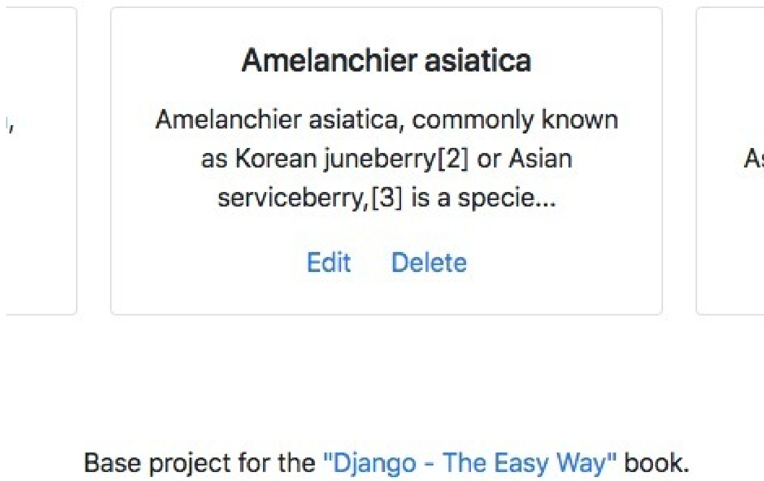
<footer class="footer"> <!-- here -->
    <div class="container">
        <span class="text-muted">
            Base project for the <a target="_blank" href="h\
ttps://leanpub.com/django-the-easy-way">"Django - The Easy \
Way" </a>book.
        </span>
    </div>
</footer>
```

Edit the base app *site.css* file and add styling for the `.footer` class:

base/static/base/css/site.css

```
.footer {
    text-align: center;
    font-size: 16px;
    height: 60px;
    line-height: 60px;
}
```

You should now see something like this:



15.5 Summary

- We now have a decent base project to work with. We use this for some of the chapters as a starting point. You might want to use this as a base for your own experiments.
- Bootstrap offers some helpful classes like `card-columns` that accomplish quite a bit with very little markup.
- Template *filters* allow you to manipulate template output like *truncate strings* or *format dates*.

16. Creating a detail page

This chapter covers

- How to add a *detail page*
- How to create *slugs*
- How to return *canonical URLS* with `get_absolute_url()`
- How to *reverse* URLS
- How to use the `{% url %}` template tag

16.1 Setup

Terminal

```
cp -fr 15-Base-Project 16-Detail-Page
cd 16-Detail-Page
source ../venv/bin/activate
```

16.2 Adding a detail page path

Edit mysite app *urls.py* file and add a path to the detail page:

mysite/urls.py

```
urlpatterns = [
    path('admin/', admin.site.urls),
    path('flower/<int:id>/', myapp_views.detail, name='deta\
il'), # < here
    path('', myapp_views.index, name='index'),
]
```

16.3 Creating the detail view

Edit myapp *views.py* file and add the *detail* function:

myapp/views.py

```
from django.shortcuts import render, get_object_or_404 # < \
here

from .models import Flower

def index(request):

    flowers = Flower.objects.all()

    return render(request, 'myapp/index.html', {'flowers': \
flowers})
```

```
def detail(request, id=None): # < here
    flower = get_object_or_404(Flower, id=id)
    return render(request, 'myapp/detail.html', {'flower': \
flower})
```

Make sure to import `get_object_or_404`.

16.4 Creating the detail page template

Create *detail.html* file in the myapp templates folder:

Detail page template

```
├── 16-Detail-Page
│   ├── myapp
│   │   ├── templates
│   │   │   └── myapp
│   │   │       ├── detail.html # < here
```

Fill it with these lines:

myapp/templates/myapp/detail.py

```
{% extends 'base/base.html' %}
{% block content %}

<div class="jumbotron">
    <div class="container">
        <h1 class="display-3">{{ flower.title }}</h1>
        <div class="lead">{{ flower.description }}</div>
    </div>
</div>

<a href="">Back<a>

{% endblock %}
```

Visit [http://127.0.0.1:8000*flower*1/](http://127.0.0.1:8000/flower/1/) and you should see the detail page *jumbotron*:



16.5 Creating slugs

Accessing individual flowers with an *id* is not the most friendly approach. Let's add a *SlugField* to hold a human-readable path.

Edit myapp *models.py* file and add a *SlugField*:

myapp/models.py

```
from django.utils.text import slugify # < here
from django.db import models

class Flower(models.Model):
    title = models.CharField(max_length=255, default='')
    description = models.TextField(default='')
    slug = models.SlugField(blank=True, default='') # < here

    def __str__(self):
        return self.title

    def save(self, *args, **kwargs): # < here
        self.slug = slugify(self.title)
        super(Flower, self).save()
```

We create the slug using the `slugify()` function in the save method.

Edit the *detail* function in the myapp views.py file and change all `id` occurrences to `slug`:

myapp/views.py

```
def detail(request, slug=None): # < here

    flower = get_object_or_404(Flower, slug=slug) # < here
```

```
    return render(request, 'myapp/detail.html', {'flower': \
flower})
```

16.6 Updating the path

Edit mysite app *urls.py* file and change the detail path:

mysite/urls.py

```
#path('flower/<int:id>/', myapp_views.detail, name='detail'\
),
path('flower/<slug:slug>/', myapp_views.detail, name='detai\
l'),
```

Run migrations:

Terminal

```
python manage.py makemigrations
python manage.py migrate
```

Edit *all* flowers you have created and save them once to generate slugs.

16.7 Defining get_absolute_url() method

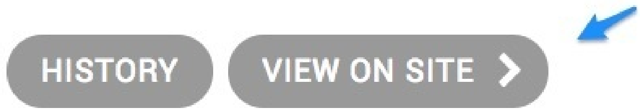
We can add a "View on site" link to the admin by defining a `get_absolute_url` method. Edit myapp *models.py* file and add the method to the *Flower* class:

myapp/models.py

```
from django.utils.text import slugify
from django.db import models
from django.urls import reverse # < here

class Flower(models.Model):
        ...
    def __str__(self):
        ...
    def save(self, *args, **kwargs):
        ...
    def get_absolute_url(self): # < here
        return reverse('detail', args=[str(self.slug)])
```

Edit a Flower object and you will see a link on the top right corner. Click it to visit the flower detail page:



16.8 Using url tag

Edit myapp *index.html* file and use the `url` tag to link the card to the detail page:

myapp/templates/myapp/index.html

```
<h5 class="card-title"><a href="{% url 'detail' flower.slug\
 %}">{{ flower.title }}</a></h5>
```

Note: make sure that each flower has a *slug* by editing and saving them once.

Visit the frontpage and click a title to see the detail page.

16.9 Details

16.9.1 Capturing URL values

You can use *angle brackets* to capture values from the URL. In here we first captured the *id* number and then the *slug*:

mysite/urls.py

```
#path('flower/<int:id>/', myapp_views.detail, name='detail'\
),
path('flower/<slug:slug>/', myapp_views.detail, name='detai\
l'),
```

You can optionally specify a *converter* type. int converter type in <int:id> means that the path matches only *integers*.

16.9.2 Using view parameters

In the myapp *views.py* file we specify a slug *parameter*. The slug from the URL will be stored in this variable. `slug=None` means that the *default* value is *None* if a parameter is not passed to this view.

myapp/views.py

```
def detail(request, slug=None):
```

`get_object_or_404` returns "*404 Page not Found*" error if the object doesn't exist. Otherwise the object with the slug from the URL parameter will be stored in the `flower` object:

myapp/views.py

```
flower = get_object_or_404(Flower, slug=slug)
```

16.9.3 Explaining slugs

Slug is a short label that contains only letters, numbers, underscores or hyphens. It's often used to offer user-friendly URLS. "*productmacbook/*" is better than "*product-113zxc/*". In our app we use the title field to create the slug.

In the myapp *models.py* we add the *SlugField* and specify `blank=True` so that the field can be empty for the `save()` method to run:

myapp/models.py

```
slug = models.SlugField(blank=True, default='')
```

Slugify function converts strings to URL slugs. You can find it in `django.utils.text`:

myapp/models.py

```
from django.utils.text import slugify
```

You can override predefined *model methods* like `save()`:

myapp/models.py

```
def save(self, *args, **kwargs):
        self.slug = slugify(self.title)
        super(Flower, self).save()
```

In the `save()` method we can make something happen when the object is saved. In this case we use it to generate a slug.

We have to call the superclass method `super()` so that the save method *default* behaviour will be executed and the object stored in the database.

`*args` and `**kwargs` allow you to collect arguments or keyword arguments and pass them to the function. This is a Python concept we don't explore in this book.

16.9.4 Reversing URLS

You can define `get_absolute_url` method to calculate a canonical URL for an object. In here we use the `reverse()` function to get the URL to a flower object:

myapp/models.py

```
def get_absolute_url(self):
        return reverse('detail', args=[str(self.slug)])
```

The `reverse` function is similar to the `url` tag that we used with the card markup. In here we pass the detail path name "detail" and the *slug* as a parameter to it.

If you have a path like this…

mysite/urls.py

```
path('flower', myapp_views.detail, name=detail),
```

… then `reverse('detail')` will generate *flower*.

If you have a path like this…

mysit/urls.py

```
path('flower/<slug:slug>/', myapp_views.detail, name='detai\
l'),
```

… then `reverse('detail', args=[str(self.slug)])` will generate a path like this *floweramelanchier-asiatica/*.

16.10 Summary

- Use angle brackets with paths to capture URL values: `'flower/<slug:slug>/'`.
- `get_object_or_404()` tries to fetch an object but returns a "Page not Found" error if the object is not found.
- *SlugField* can be used to store a user-friendly path.
- It's useful to define the `get_absolute_url()` method for a model to have an easy access to canonical URLS.
- Use `{% url %}` tag or `{{ object.get_absolute_url }}` in templates instead of hardcoding URLS.

17. Adding category as a many-to-one relationhip

This chapter covers

- Many-to-one relationships with *ForeignKey*
- How to access *related* objects

17.1 Setup

Terminal

```
cp -fr 15-Base-Project 17-Category-ManyToOne
cd 17-Category-ManyToOne
source ../venv/bin/activate
```

17.2 Adding category field and model

Edit myapp *models.py* file and add a *Category* class and a *category* field:

myapp/models.py

```
from django.db import models

class Category(models.Model): # < here

    title = models.CharField(max_length=255, default='')

    def __str__(self):
        return self.title

class Flower(models.Model):

    title = models.CharField(max_length=255, default='')
        description = models.TextField(default='')
        category = models.ForeignKey(Category, null=True, on_delet\
e=models.PROTECT) # < here

    def __str__(self):
        return self.title
```

Edit myapp *admin.py* and register the *Category* model:

myapp/admin.py

```
from django.contrib import admin

from myapp.models import Flower, Category # < here
```

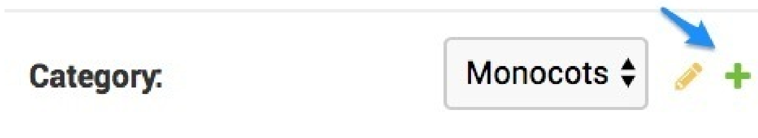
```
admin.site.register(Flower)
admin.site.register(Category) # < here
```

Run migrations:

Terminal

```
python manage.py makemigrations
python manage.py migrate
```

Edit the flowers and select a category for each item. You can create the referenced *Category* object while you are editing the Flower objects:



17.3 Updating the homepage template

Edit the myapp *index.html* template file and print out the category:

myapp/templates/myapp/index.html

```
<p class="card-text">{{ flower.description | truncatechars:\
100 }}</p>
<a href="#" class="card-link">{{ flower.category }}</a> <!-\
- here -->
```



17.4 Details

17.4.1 Examining many-to-one relationships

ForeignKey is a *many-to-one* relationship:

myapp/models.py

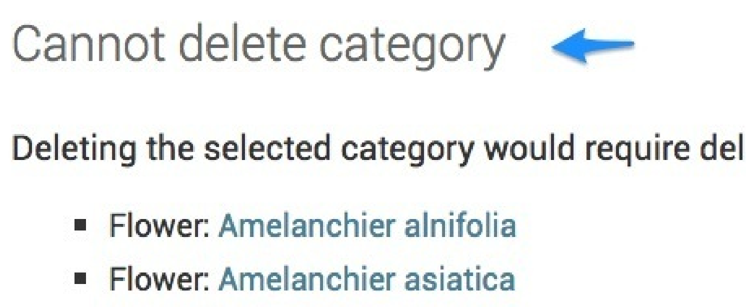
```
category = models.ForeignKey(Category, on_delete=models.PRO\
TECT, null=True)
```

Categories can link to many flowers but each flower can have a reference to only one category.

`ForeignKey` field requires two arguments: the related model *class* and `on_delete` option.

The *Flower* model is related to *Category* class so we specify that as the first argument.

`on_delete=models.PROTECT` prevents the deletion of a *Category* object if it’s referenced by a *Flower* object:



You can delete categories that are not referenced by any flower.

`null=True` means that an empty field will be stored as *NULL* in the database. This allows us to run the initial migration without specifying a default value.

17.4.2 Accessing related objects

You can access related objects the same way you access any attribute:

Dot notation

```
{{ flower.category }}
```

If you need to get all *flowers* that link to a specific category, you can use `_set` like this:

Get related flowers

```
{{ category.flower_set }}
```

You can test this by adding the following code inside the *card* div in the myapp *index.html* file:

myapp/templates/myapp/index.html

```
<div class="card">
        ...
    <hr>

    All flowers in the <strong>{{ flower.category }}</stron\
g> category:<br>
    {% for c_flower in flower.category.flower_set.all %}
      <a href="#" class="card-link">{{ c_flower }}</a><br>
    {% endfor %}

</div>
```

Use `all` in `flower.category.flower_set.all` so you have an *iterable* to loop through.



17.5 Summary

- *ForeignKey* is a *many-to-one* relationship. Another example would be a car model that has a foreignkey relationship to a *brand* model. Each car object

can link to only one *brand* object like “Audi” or “Mercedes-Benz” but the brands can link to many car objects.

- Make sure to *register* the *Category* model in the *admin.py* file so you can create the referenced objects on the fly.
- If you set `null=True` for a field, empty values will be stored as *NULL* in the database.

18. Referencing tags with a ManyToMany field

This chapter covers

- How to reference multiple items with many-to-many relationships

18.1 Setup

Terminal

```
cp -fr 15-Base-Project 18-Tags-ManyToMany
cd 18-Tags-ManyToMany
source ../venv/bin/activate
```

18.2 Adding the tags field

Edit myapp *models.py* file and add `Tag` model and `tags` field:

myapp/models.py

```
from django.db import models

class Tag(models.Model): # < here

    title = models.CharField(max_length=255, default='')

    def __str__(self):
        return self.title

class Flower(models.Model):

    title = models.CharField(max_length=255, default='')
    description = models.TextField(default='')
    tags = models.ManyToManyField(Tag) # < here

    def __str__(self):
        return self.title
```

Edit myapp *admin.py* file and register the `Tag` model:

myapp/admin.py

```
from django.contrib import admin

from myapp.models import Flower, Tag # < here

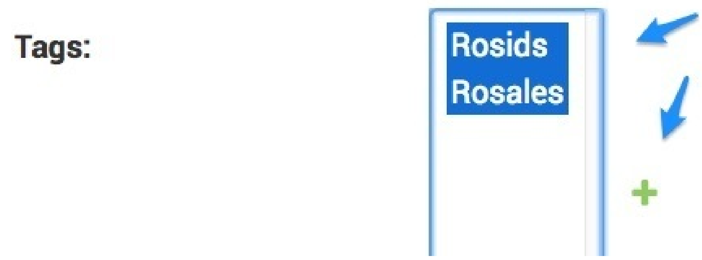
admin.site.register(Flower)
admin.site.register(Tag) # < here
```

Run migrations:

Terminal

```
python manage.py makemigrations
python manage.py migrate
```

Edit a flower and add some tags. Make sure to *select* more than one tag:

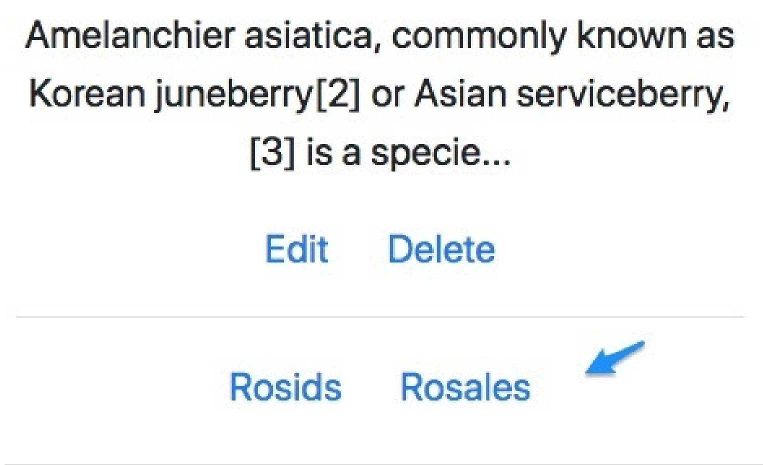


18.3 Updating the homepage template

Edit the myapp *index.html* template file and print out the tags:

myapp/templates/myapp/index.html

```
<div class="card">
        ...
    <hr>
    {% for tag in flower.tags.all %}
        <a href="#" class="card-link">{{ tag }}</a>
    {% endfor %}
</div>
```



18.4 Summary

- *ManyToMany* relationship allows our flowers to reference many tags and the tags to reference many flowers.

19. Creating a tags page

This chapter covers

- How to create a “tags” page to display *tagged* items
- How to do *lookups* across relationships
- How to *re-use templates*

19.1 Setup

Terminal

```
cp -fr 18-Tags-ManyToMany 19-Tags-Page
cd 19-Tags-Page
source ../venv/bin/activate
```

19.2 Adding tags path

Edit mysite *urls.py* file and add a path to the tags page:

mysite/urls.py

```
urlpatterns = [
    path('admin/', admin.site.urls),
    path('', myapp_views.index, name='index'),
    path('tags/<slug:slug>/', myapp_views.tags, name='tags'\
), # < here
]
```

19.3 Adding the slug field

Edit myapp *models.py* file and add a *SlugField* to the *Tag* model:

myapp/models.py

```
from django.db import models
from django.utils.text import slugify # < here

class Tag(models.Model):

    title = models.CharField(max_length=255, default='')
    slug = models.SlugField(blank=True, default='') # < here

    def __str__(self):
        return self.title

    def save(self, *args, **kwargs): # < here
        self.slug = slugify(self.title)
        super(Tag, self).save()
```

19.4 Creating the tags view

Edit myapp *views.py* file and add a *tags* view function:

myapp/views.py

```
from django.shortcuts import render

from myapp.models import Flower

def index(request):

    flowers = Flower.objects.all()

    return render(request, 'myapp/index.html', {'flowers': \
flowers })

def tags(request, slug=None): # < here

    flowers = Flower.objects.filter(tags__slug=slug)

    return render(request, 'myapp/index.html', {'flowers': \
flowers })
```

Run migrations:

Terminal

```
python manage.py makemigrations
python manage.py migrate
```

Visit *adminmyapp/tag/*. Edit and save the tag objects to generate slugs.

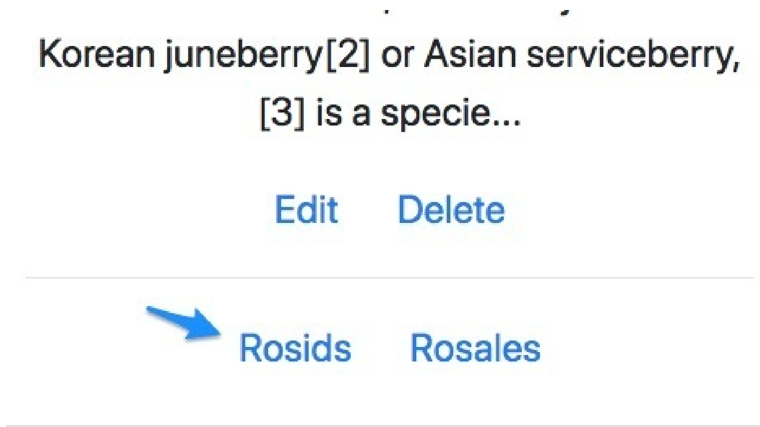
19.5 Updating homepage template

Edit myapp *index.html* file and use `{% url 'tags' tag.slug %}` to generate the link:

myapp/templates/myapp/index.html

```
<hr>
{% for tag in flower.tags.all %}
        <a href="{% url 'tags' tag.slug %}" class="card-link">{{ t\
ag }}</a> <!-- here -->
{% endfor %}
```

Now the frontpage tags link to the tags page:



Click the tag links and you will see the according tag page: `tagsrosales/`. If you have Flowers tagged with “Rosales”, you will only see those items in this page:



19.6 Details

19.6.1 Doing lookups across relationships

In myapp *views.py* file we fetch objects that are tagged with a specific tag:

myapp/views.py

```
def tags(request, slug=None):

    flowers = Flower.objects.filter(tags__slug=slug) # < he\
re

    return render(request, 'myapp/index.html', {'flowers': \
flowers })
```

With `filter` function you can return a *QuerySet* that match lookup parameters. In this case our parameter is `tags__slug=slug`. This will return all *flower* objects that has a reference to a *tag* object with the *slug* from the URL. `tagsrosales/` would fetch all flowers tagged with “Rosales”.

Django has plenty of other query interaction tools. See https://samuli.to/QuerySet-API.

19.6.2 Reusing templates

You might have noticed that we are using the same myapp *index.html* in the frontpage and in the tags page. Reusing templates will save you a lot of time and makes it easier to make changes. Now if we want to change the card styling or markup, we can do it in one place. The changes will show up in the frontpage *and* in the tags page.

19.7 Summary

- Django offers a big selection of methods like `filter()` to modify your data queries.
- You can do *lookups through relationships* using the double underscore (__) syntax: `tags__slug=slug`.
- *Reusing templates* will make your app look consistent and easier to maintain.

20. Creating a search feature

This chapter covers

- How to create a simple search feature
- How to work with GET parameters

20.1 Setup

Terminal

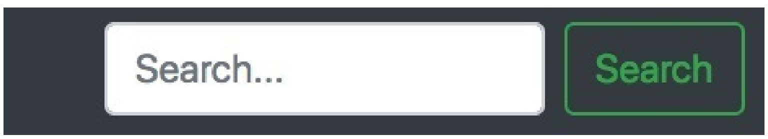
```
cp -fr 18-Tags-ManyToMany 20-Search
cd 20-Search
source ../venv/bin/activate
```

20.2 Adding a search form

Edit *base.html* file and add the following `<form>` element at the bottom of the `<nav>` element:

base/templates/base/base.html

```
<nav>
..
    <form action="/" method="get" class="form-inline mt-2 m\
t-md-0">
        <input id="q" name="q" value="{{ request.GET.q }}" \
class="form-control mr-sm-2" type="text" placeholder="Searc\
h..." aria-label="Search">
        <button class="btn btn-outline-success my-2 my-sm-0\
" type="submit">Search</button>
    </form>
</nav>
```



20.3 Updating the index view

Edit the myapp *views.py* file and replace the contents with these lines:

myapp/views.py

```
from django.shortcuts import render

from myapp.models import Flower

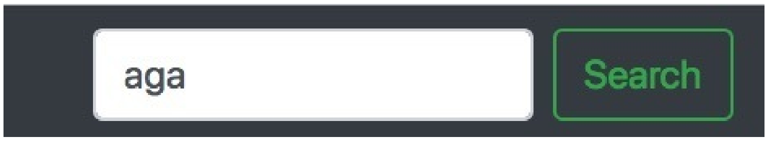
def index(request):

    q = request.GET.get('q', None)
    items = ''
    if q is None or q is "":
        flowers = Flower.objects.all()
    elif q is not None:
        flowers = Flower.objects.filter(title__contains=q)

    return render(request, 'myapp/index.html', {'flowers': \
flowers })
```

Now you can search titles by providing a *q* GET parameter in the URL:

http://127.0.0.1:8000/?q=aga



We are again using the same *index.html* template:



20.4 Details

When a user requests a page like our frontpage, Django creates an *HttpRequest* object. This object contains *metadata* about that request. This includes all *GET* parameters.

We can then access those parameters in *HttpRequest.GET*. In this case we only send one, the *q* parameter. This is then used in the myapp index view.

If we don't provide the *q* parameter or it is an empty string, then all objects are fetched: `flowers = Flower.objects.all()`.

If *q* is provided, we fetch all flowers where the title field contains the query string: `Flower.objects.filter(title__contains=q)`.

20.5 Summary

- *Bootstrap* provides a generic template that you can use for the search form.
- *HttpRequest* object contains metadata about a *request*. We can act on that data inside views. Like filter items based on a GET parameter.

21. Working with forms: creating items

This chapter covers

- How to create forms with ModelForm

21.1 Setup

Terminal

```
cp -fr 15-Base-Project 21-Forms-Create
cd 21-Forms-Create
source ../venv/bin/activate
```

21.2 Creating the edit form

Create an *edit.html* file in the myapp templates folder:

Template location

```
├── myapp
│   ├── templates
│       └── myapp
│           ├── edit.html # < here
│           ├── index.html
```

Fill it with these lines:

myapp/templates/myapp/edit.html

```
{% extends 'base/base.html' %}
{% block content %}

<form action="" method="post">
    {% csrf_token %}
    <div class="row justify-content-center">
      <div class="col-6">
        {{ form }}
        <hr class="mb-3">
        <button class="btn btn-primary btn-lg btn-block" ty\
pe="submit">Submit</button>
      </div>
    </div>
</form>

{% endblock %}
```

We will use this template to *create* and *edit* flower items.

21.3 Creating the form class

Create *forms.py* file in the myapp folder:

forms.py location

```
├── myapp
        ..
│   ├── admin.py
│   ├── apps.py
│   ├── forms.py # < here
```

Fill it with these lines:

myapp/forms.py

```
from django import forms
from django.forms import ModelForm
from .models import Flower

class MyForm(ModelForm):
    title = forms.CharField(label='Title',
        widget= forms.TextInput(attrs={'class': 'form-contr\
ol '}))
    class Meta:
        model = Flower
        fields = ['title']
```

21.4 Updating urlpatterns

Edit mysite app *urls.py* file and add the *create* path:

mysite/urls.py

```
urlpatterns = [
    path('admin/', admin.site.urls),
    path('', myapp_views.index, name='index'),
    path('flower/create/', myapp_views.create, name='create\
'), # < here
]
```

21.5 Creating the view function

Edit myapp *views.py* file and add a *create* view below the *index* view:

myapp/views.py

```
from django.shortcuts import render
from .models import Flower
from django.http import HttpResponseRedirect # < here
from .forms import MyForm # < here

def index(request):
        ...
def create(request): # < here
    if request.method == 'POST':
        form = MyForm(request.POST)
        if form.is_valid():
```

```
            form.save()
            return HttpResponseRedirect('/')
    else:
        form = MyForm()
    return render(request, 'myapp/edit.html', {'form': form\
})
```

21.6 Adding a menu item

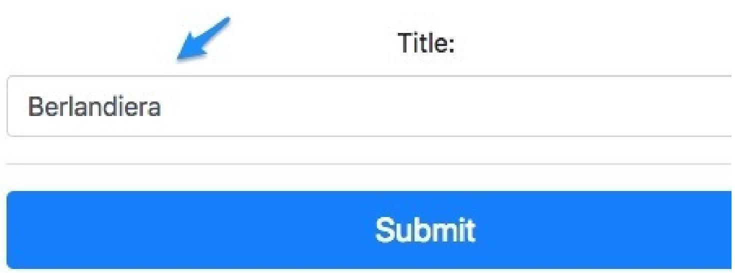
Edit base app *base.html* file and add a menu link to the flower creation form:

base/templates/base/base.html

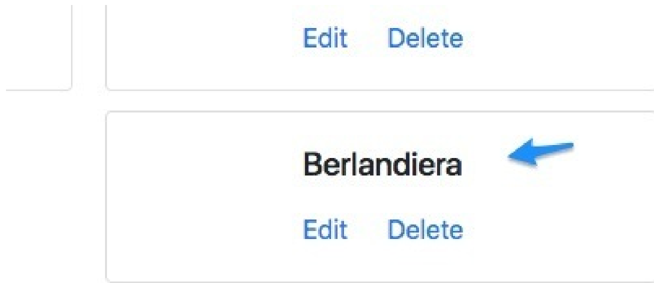
```
<ul>
    <li><a>Home</a></li>
    <li class="nav-item"> <!-- here -->
        <a class="nav-link" href="flowercreate/">
        Create Flower
        </a>
    </li>
</ul>
```

I removed unimportant CSS classes for the book. The complete markup is available at the GitHub repository.

Visit *flowercreate/* and create a flower:



The new flower will now show up on the frontpage:



Note that the bootstrap class `card-columns` creates a *masonry* like arrangement, not a grid.

21.7 Details

21.7.1 Protecting against cross site request forgeries

In the myapp *edit.html* file we define a *CSRF* token:

myapp/templates/myapp/edit.html

```
<form action="" method="post">
    {% csrf_token %} # < here
        ...
</form>
```

This token adds protection against *Cross Site Request Forgeries* where malicious parties can cause visitor's browser to make a request to your website. The cookies in the visitor browser make the app think that the request came from an authorized source.

Use the token only in *POST* requests. You don't need it with *GET* requests. Any request that has a potential to change the system shoud be a *POST* request. Like when we add flowers to the database.

GET requests are often used in situations where the system state is not changed, like when we query database with the search form. The *q* search word parameter is public data we don't need to hide. You want to be able to share links like this: https://samulinatri.com/search?q=Django.

Also you shouldn't use the token with forms that point to *external* URLS. This introduces a vulnerability as the token is leaked. `action=""` in the form means

that the *POST* data is sent to the *current* internal URL (`flowercreate/`).

21.7.2 Adding form fields

Easiest way to generate HTML markup for the form fields is to use the `{{ form }}` template variable:

myapp/templates/myapp/edit.html

```
<div class="col-6">
    {{ form }}
</div>
```

This will produce the following HTML:

Generated HTML

```
<div class="col-6">
    <label for="id_title">Title:</label>
    <input type="text" name="title" maxlength="255" class="\
form-control" required="" id="id_title">
</div>
```

21.7.3 Using the Form class

Form class represents a form. It describes a form in a similar way the `Flower` model describes how fields map to *database fields*. With forms the fields map to *HTML elements*.

ModelForm is a helper class that creates that *Form* class from a *Model*:

myapp/forms.py

```
class MyForm(ModelForm):
    title = forms.CharField(label='Title',
        widget= forms.TextInput(attrs={'class': 'form-contr\
ol '}))
    class Meta:
        model = Flower
        fields = ['title']
```

With *ModelForm* we don’t need to specify the fields *again*. We already add the fields in the Flower model:

Fields are already specified in the models.py file

```
class Flower(models.Model):
    title = models.CharField(max_length=255, default='')
    description = models.TextField(default='')
```

This would be enough to create a form to edit *all* Flower fields:

myapp/forms.py

myapp/forms.py

```
class MyForm(ModelForm):
    class Meta:
        model = Flower
        fields = '__all__' # < here
```

It's recommended to *explicitly* specify all the fields like this though:

Fields should be explicitly specified

```
fields = ['title', 'description']
```

Otherwise you could unintentionally expose fields to users when you add them to the model.

A form field is represented as an HTML "widget" that produces some default markup. We can modify that widget in the form definition:

Adding CSS class for Bootstrap

```
title = forms.CharField(label='Title',
        widget = forms.TextInput(attrs={'class': 'form-cont\
rol '}))
```

The only reason we did this is because we wanted to add the `form-control` CSS class to the title input element. This way we can take advantage of the Bootstrap textual form control styling.

21.7.4 Examining the view function

In the myapp *views.py* file we added the *create* view function:

myapp/views.py

```
def create(request):
    if request.method == 'POST':
        form = MyForm(request.POST)
        if form.is_valid():
            form.save()
            return HttpResponseRedirect('/')
    else:
        form = MyForm()
    return render(request, 'myapp/edit.html', {'form': form\
})
```

First we check if the request is *POST*. If it's *not*, we create an *empty* form that we pass to the *edit.html* template:

Empty form is passed to the template

```
if request.method == 'POST':
    ..
else
```

```
    form = MyForm()
return render(request, 'myapp/edit.html', {'form': form})
```

This is the default scenario when you *first* visit the *flowercreate/* page. We need to create the `form` object so that the form HTML can be generated using the template tags.

If the request *is* POST, we create the `form` object and populate it with the data from the *request*:

Populating the form object with the POST data

```
if request.method == 'POST':
    form = MyForm(request.POST)
```

Then we check if the form data is *valid* and *save* the flower:

Validating and saving the data

```
if form.is_valid():
    form.save()
    return HttpResponseRedirect('/')
```

Django has built-in *validators* that it uses internally. For example `EmailValidator` for email addresses and `validate_slug` for slugs. If the input doesn't satisfy the validator, a *ValidationError* is raised.

The `save()` method creates the flower object from the data *bound* to the form and stores it in the database.

When we submit a form using a *POST* request, our *create* view will instantiate the form object and populate it with the form data from the request. We "bind" the data to the form. It's now a "bound" form.

The validated data can be accessed in the `form.cleaned_data` dictionary:

Accessing validated data

```
if form.is_valid():
    print(form.cleaned_data['title']) # < here
    form.save()
    return HttpResponseRedirect('/')
```

This will print the validated title field data in the terminal:



And finally `HttpResponseRedirect('/')` redirects the visitor to the frontpage.

21.8 Summary

- Use `{% csrf_token %}` with *internal POST* forms to protect against *Cross Site Request Forgeries*.
- `{{ form }}` template variable generates markup for all form fields.
- `Form` class represents a form. Its fields map to HTML elements.
- `ModelForm` is a helper class that allows us create the `Form` class from a Django model.
- A form field is represented as an HTML “widget”. You can modify this widget in the form definition.
- The submitted form is *processed* in the *create* view.
- Django has *built-in* validation that triggers a *ValidationError* when the data doesn’t validate.
- *validated* data is stored in the `form.cleaned_data` dictionary.
- In the *create* view we *bind* the form data to the *form* instance.
- `form.save()` method creates a database object using the bound data.

22. Working with forms: editing items

This chapter covers

- How to create an *edit* form
- *Primary key* and *id* field

22.1 Setup

Terminal

```
cp -fr 21-Forms-Create 22-Forms-Edit
cd 22-Forms-Edit
source ../venv/bin/activate
```

22.2 Adding the path

Edit mysite app *urls.py* file and add the *edit* path:

mysite/urls.py

```
urlpatterns = [
    path('admin/', admin.site.urls),
    path('', myapp_views.index, name='index'),
    path('flower/create/', myapp_views.create, name='create\
'),
    path('flower/edit/<int:pk>/', myapp_views.edit, name='e\
dit'), # < here
]
```

22.3 Creating the edit view

Edit myapp *views.py* file and add the *edit* view function:

myapp/views.py

```
from django.shortcuts import render, get_object_or_404 # < \
here
from .models import Flower
from django.http import HttpResponseRedirect
from .forms import MyForm

def index(request):
        ...
def create(request):
        ...
def edit(request, pk=None): # < here
    flower = get_object_or_404(Flower, pk=pk)
    if request.method == "POST":
        form = MyForm(request.POST, instance=flower)
```

```
        if form.is_valid():
            form.save()
            return HttpResponseRedirect('/')
    else:
        form = MyForm(instance=flower)

    return render(request, 'myapp/edit.html', {'form': form\
})
```

22.4 Updating the edit link

Edit myapp *index.html* file and change the *edit* link to this:

myapp/templates/myapp/index.html

```
<a href="{% url 'edit' pk=flower.pk %}" class="card-link">E\
dit</a>
```

You can now edit flowers by clicking the *Edit* links on the frontpage.



22.5 Details

22.5.1 Capturing the id

In the *edit* path we *capture* the flower *id*:

Edit path

```
path('flower/edit/<int:pk>/', myapp_views.edit, name='edit'\
),
```

"*pk*" is a *shortcut* to the model *primary key*. "*id*" is the name of the default *primary key* field. Take a look at the *0001_initial.py* file in the myapp migrations

folder:

Django creates the id field automatically

```
...
fields=[
        ('id', models.AutoField..), # < here
        ('title', models.CharFi..)],
...
```

Django will automatically add the *id AutoField* if you don't specify `primary_key=True` on any of the fields.

It's more flexible to use the `flower.pk` shortcut when accessing the id field. This way you can use the same code to access the id even if you *change* the primary key field.

22.5.2 Examining the edit view

In myapp *views.py* file we add the *edit* view function. It is very much like the *create* view function but with a few changes:

Edit view is almost like the create view

```
def edit(request, pk=None): # < new
    flower = get_object_or_404(Flower, pk=pk) # < new
    if request.method == "POST":
        form = MyForm(request.POST, instance=flower) # < new
        if form.is_valid():
            form.save()
            return HttpResponseRedirect('/')
    else:
        form = MyForm(instance=flower) # < new

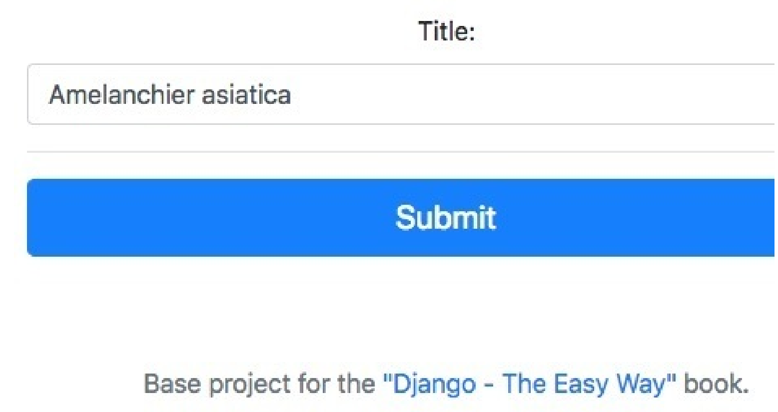
    return render(request, 'myapp/edit.html', {'form': form\
})
```

First we pass the captured *pk* to the view with `pk=None`. *None* is the default value if *pk* argument is not provided.

`get_object_or_404` raises an *Http404* exception and returns a standard *404 (page not found)* error page if the object matching the lookup parameters (*pk=pk*) is not found.

MyForm inherits from *ModelForm* that can accept a model *instance* as a keyword argument. This means that the `form.save()` method will now *update* an existing flower instead of creating a new one.

We also use it to populate the *initial* form with form = MyForm(instance=flower). When you visit *floweredit/<pk>/* you will be able to *see* and *edit* the existing data:



22.6 Summary

- *pk* is a shortcut to the model *primary key* field. Django creates a default *id* field automatically unless you set the primary key on any field with primary_key=True.
- get_object_or_404 fetches an object or returns a *page not found* view if it can't find the object matching the lookup parameters.
- *instance* keyword argument allows us to *update* an existing object with form.save() method and populate the form with an existing data for editing.

23. Working with forms: customization

This chapter covers

- How to change the *order* of the fields
- How to render *validation* errors manually

23.1 Setup

Terminal

```
cp -fr 22-Forms-Edit 23-Forms-Customization
cd 23-Forms-Customization
source ../venv/bin/activate
```

23.2 Adding the description field

If you want to have more control for the form markup, you can print out the form fields manually. Let's add a *description* field to the form and customize the template.

Edit myapp *forms.py* file and add the *description* field to the fields list:

myapp/forms

```
from django import forms
from django.forms import ModelForm
from .models import Flower

class MyForm(ModelForm):
    title = forms.CharField(label='Title',
        widget = forms.TextInput(attrs={'class': 'form-cont\
rol '}))
    description = forms.CharField(label='Description', # < \
here
        widget = forms.Textarea(attrs={'class': 'form-contr\
ol '}))
    class Meta:
        model = Flower
        fields = ['title', 'description'] # < here
```

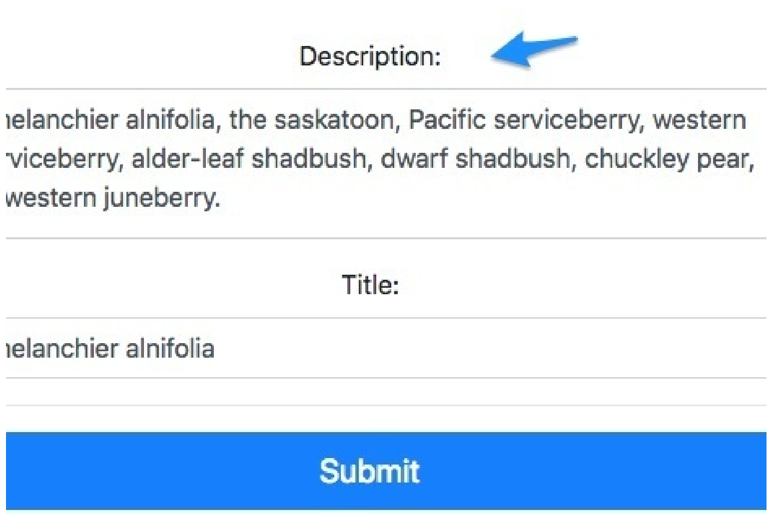
Edit myapp *edit.html* template and replace the `{{ form }}` template variable with these lines:

myapp/templates/myapp/edit.html

```
{{ form.non_field_errors }}
```

```
<div class="form-group">
    {{ form.description.errors }}
    {{ form.description.label_tag }}
    {{ form.description }}
</div>

<div class="form-group">
    {{ form.title.errors }}
    {{ form.title.label_tag }}
    {{ form.title }}
</div>
```



23.3 Details

23.3.1 Changing field order

If you just need to change the order of the fields, you can do it in the myapp *forms.py* file:

Update fields list to change order

```
class Meta:
    model = Flower
    fields = ['description', 'title'] # < here
```

If you need more flexibility, edit the myapp *edit.html* template and print the form fields manually.

23.3.2 Customizing validation errors

Inputing invalid data generates a validation error. Use `{{ form.title.errors }}` to display those errors manually.

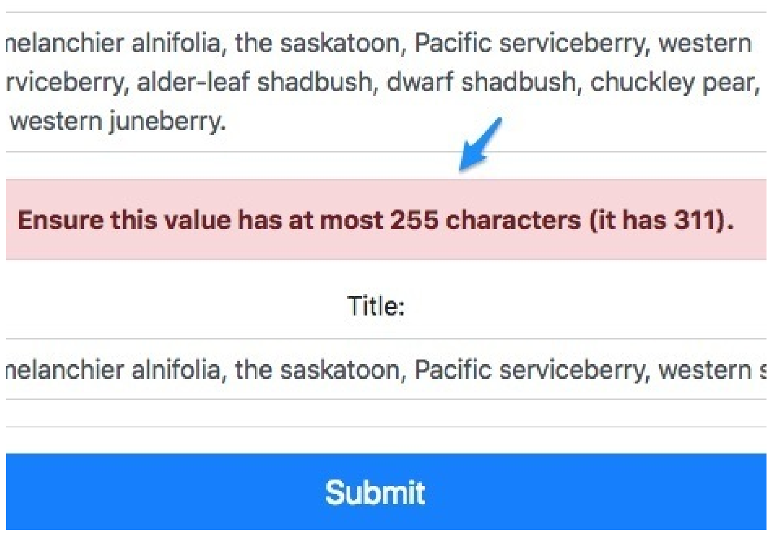
`{{ form.non_field_errors }}` will render non-field specific general errors.

Note that `{{ form }}` renders all fields *with the errors*.

You could go even further and loop through the errors manually. Replace `{{ form.title.errors }}` with these lines:

Looping through errors manually

```
{% if form.title.errors %}
    <ol class="alert alert-danger">
    {% for error in form.title.errors %}
        <li><strong>{{ error|escape }}</strong></li>
    {% endfor %}
    </ol>
{% endif %}
```



Check out the official documentation for more theming options: https://samuli.to/Form-Templates

23.4 Summary

- You can change the form field order in the form definition: `fields = ['description', 'title']`.
- `{{ form }}` renders all markup for the fields you specified in the form class. Including the *errors*.

- For more control, you can use `{{ form.title.errors }}`, `{{ form.title.label_tag }}` and `{{ form.title }}` to render the form markup *manually*.

24. Creating and deleting objects

This chapter covers

- How to *delete* Flower objects with a custom view
- How to use the *Python interactive interpreter* to manipulate objects and interact with Django

24.1 Setup

Terminal

```
cp -fr 23-Forms-Customization 24-Object-Manipulation
cd 24-Object-Manipulation
source ../venv/bin/activate
```

24.2 Adding the delete path

Edit mysite *urls.py* file and add the *delete* path:

mysite/urls.py

```
urlpatterns = [
    path('admin/', admin.site.urls),
    path('', myapp_views.index, name='index'),
    path('flower/create/', myapp_views.create, name='create\
'),
    path('flower/edit/<int:pk>/', myapp_views.edit, name='e\
dit'),
    path('flower/delete/<int:pk>/', myapp_views.delete, nam\
e='delete'), # < here
]
```

24.3 Adding the delete view

We don't necessary need a *form* to delete items. You could simple capture the *pk* from the URL and do the deletion logic in a view.

Edit myapp *views.py* file and add the *delete* view:

myapp/views.py

```
def index(request):
        ...
def create(request):
        ...
def edit(request, pk=None):
        ...
```

```python
def delete(request, pk=None): # < here
    flower = get_object_or_404(Flower, pk=pk)
    flower.delete()

    return render(request, 'myapp/index.html')
```

24.4 Updating the delete link

Edit the myapp *index.html* template and update the delete link:

myapp/templates/myapp/index.html

```
<div class="card-body">
        ...
        <a href="{% url 'edit' pk=flower.pk %}" class="card-link">\
Edit</a>
        <a href="{% url 'delete' pk=flower.pk %}" class="card-link\
">Delete</a> # < here
</div>
```

You can now use the delete links on the homepage to erase items.

24.5 Details

Make sure you have activated the virtual environment and open the *Python interactive interpreter*:

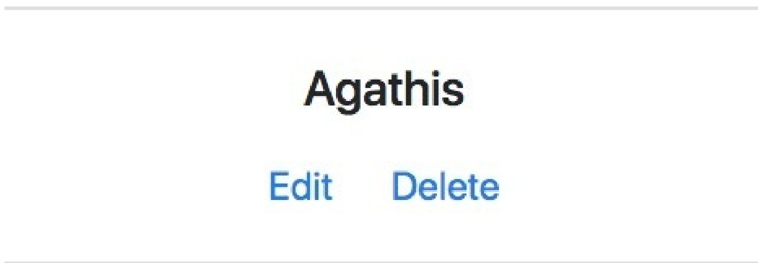
Interactive interpreter

```
python manage.py shell
>>> from myapp.models import Flower
>>> flower = Flower(title="Agathis")
>>> flower
<Flower: Agathis>
>>> flower.save()
```

`python manage.py shell` starts the interactive session.

Flower model can be *instantiated* like any class. `Flower(title="Agathis")` creates a new Flower object with the title “Agathis”.

`Flower.save()` stores it in the database. Visit homepage to confirm that it was actually created:



In the myapp *views.py* file we use `flower.delete()` method to delete the object from the database:

delete() method erases the object from the database

```
flower = get_object_or_404(Flower, pk=pk)
flower.delete()
```

You can do the same thing in the interactive interpreter:

Interactive interpreter

```
>>> flower.delete()
(1, {'myapp.Flower': 1})
>>>
```

`flower.delete()` returns how many objects were *deleted* and how many deletions were executed by object type: `{'myapp.Flower': 1}`. We deleted *1* object of the type *Flower*.

You can *get* and *update* an object like this:

Interactive interpreter

```
>>> flower = Flower.objects.get(pk=1)
>>> flower
<Flower: Amelanchier alnifolia...>
>>> flower.title = "UPDATED"
>>> flower.save()
>>> flower
<Flower: UPDATED>
>>>
```



24.6 Summary

- You can use the *Python interactive interpreter* to run Python code and interact with your Django apps.
- `object = Class()` instantiates a *Class* object.
- `object.save()` saves the object to the database or updates it.
- `object.delete()` deletes the object from the database.

25. Authenticating users with Allauth

This chapter covers

- How to create a complete authentication system with Allauth
- How to use Bootstrap 4 with the default templates

25.1 Setup

Terminal

```
cp -fr 15-Base-Project 25-Authentication
cd 25-Authentication
source ../venv/bin/activate
```

25.2 Installing Allauth

Install the *Allauth* package:

Terminal

```
pip install django-allauth
```

Update the *settings.py* file:

mysite/settings.py

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'django.contrib.sites', # < here
    'allauth', # < here
    'allauth.account', # < here
    'allauth.socialaccount', # < here
    'base',
    'myapp',
]

SITE_ID = 1 # < here
EMAIL_BACKEND = 'django.core.mail.backends.console.EmailBac\
kend' # < here
LOGIN_REDIRECT_URL = '/' # < here
```

Add *accounts* path to the *urls.py* file:

mysite/urls.py

mysite/urls.py

```
from django.contrib import admin
from django.urls import path, include # < here
from myapp import views as myapp_views

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', myapp_views.index, name='index'),
    path('accounts/', include('allauth.urls')), # < here
]
```

Run migrations:

Terminal

```
python manage.py migrate
```

Open another browser or logout and create a *test* account in *accountssignup/*:



25.3 Creating template files

Edit mysite app *settings.py* file and add the templates folder to the `'DIRS': [],` list:

Locating templates

```
'DIRS': [os.path.join(BASE_DIR, 'templates'), os.path.join(\
BASE_DIR, 'templates', 'allauth')],
```

Create a *templates* folder in the root of the site. Create *allauth* folder inside it. Copy the *account* folder from the allauth package folder inside it:

Terminal

```
mkdir templates
cd templates
mkdir allauth
cd allauth
cp -fr ../../../venv/lib/python3.7/site-packages/allauth/te\
mplates/account .
```

The folder structure should now look like this:

Allauth templates

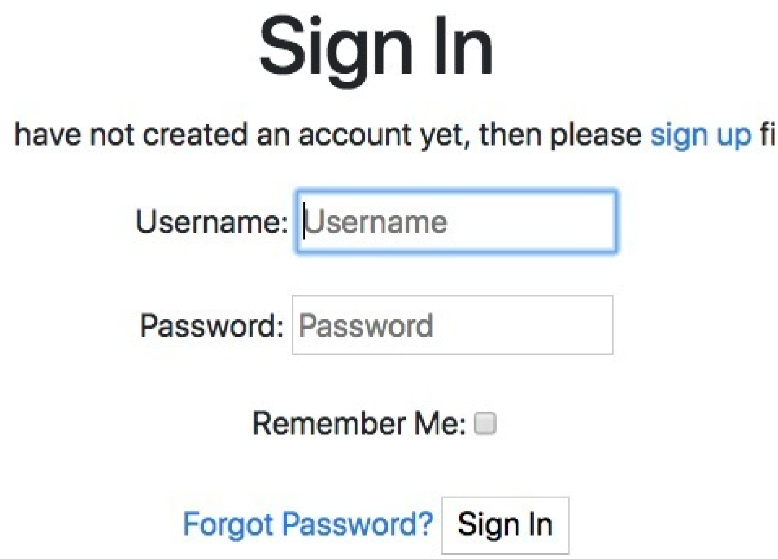
```
.
├── base
├── db.sqlite3
├── manage.py
├── myapp
├── mysite
└── templates
    └── allauth
        └── account
                        ...
            ├── base.html
            ├── login.html
            ├── logout.html
                        ...
```

Change the *base.html* contents in the *account* folder to this:

templates/allauth/account/base.html

```
{% extends "base/base.html" %}
```

Logout in *accounts/logout/* and visit *accountssignin/*. You should see the login form wrapped inside the base theme:



25.4 Updating the templates for Bootstrap 4

Install *django-widget-tweaks* package:

Terminal

```
pip install django-widget-tweaks
```

Add `widget_tweaks` to the `INSTALLED_APPS` list:

mysite/settings.py

```
INSTALLED_APPS = [
        ...
    'allauth',
    'allauth.account',
    'allauth.socialaccount',
    'widget_tweaks', # < here
    'base',
    'myapp',
]
```

Create a *form_snippet.html* inside the root *templates* folder:

templates/form_snippet.html

```
{% load widget_tweaks %}

{% for field in form %}
<div class="fieldWrapper mb-1">
  {{ field.errors }}
  {% if field.field.widget.input_type != 'checkbox' %}
    <label class="sr-only" for="{{ form_field.auto_id }}">{\
```

```
{ form_field.label }}</label>
    {{ field|add_class:"form-control" }}
  {% else %}
    {{ field.label_tag }}
    {{ field }}
  {% endif %}
</div>
{% endfor %}
```

We can now re-use this snippet to render all fields in any template.

Edit *login.html* file in the templates *allauth/account/* folder. Replace the *form* element with these lines:

templates/allauth/account/login.html

```
<form class="form-account login" method="POST" action="{% u\
rl 'account_login' %}">

  {% csrf_token %}

  {% include 'form_snippet.html' %} <!-- here -->

  {% if redirect_field_value %}
    <input type="hidden" name="{{ redirect_field_name }}" v\
alue="{{ redirect_field_value }}" />
  {% endif %}

  <a class="button secondaryAction d-block mb-2" href="{% u\
rl 'account_reset_password' %}">{% trans "Forgot Password?"\
 %}</a>
  <button class="btn btn-lg btn-primary btn-block" type="su\
bmit">{% trans "Sign In" %}</button>

</form>
```

Notice the form element `form-account` CSS class. Add the form styling in *site.css*:

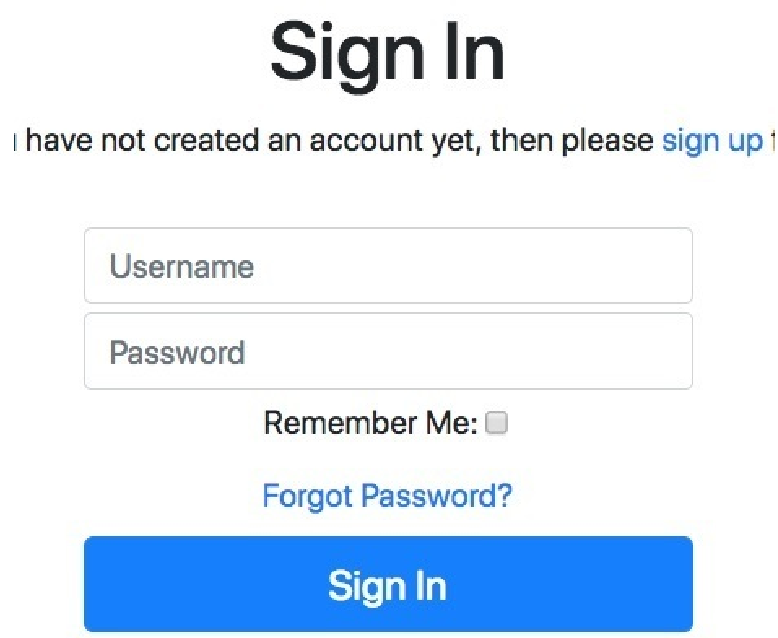
base/static/base/css/site.css

```
body {
    padding-top: 5rem;
}
.starter-template {
    padding: 3rem 1.5rem;
    text-align: center;
}
.footer {
    text-align: center;
    font-size: 16px;
    height: 60px;
    line-height: 60px;
}

.form-account { // < here
    width: 100%;
    max-width: 330px;
    padding: 15px;
```

```
    margin: auto;
}
```

Visit *accountslogin/* and you should see this:



Edit *signup.html* file in the templates *allauth/account/* folder. Replace the *form* element with this:

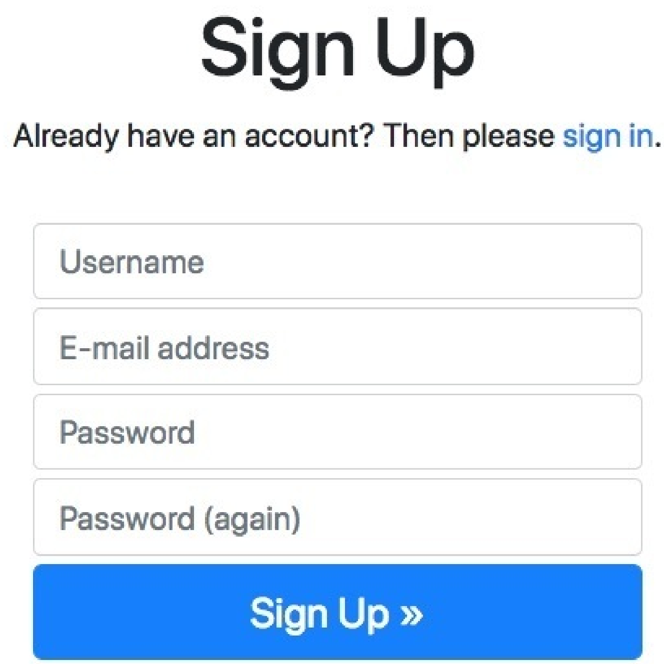
templates/allauth/account/signup.html

```
<form class="form-account signup" id="signup_form" method="\
post" action="{% url 'account_signup' %}">
  {% csrf_token %}

  {% include 'form_snippet.html' %}

  {% if redirect_field_value %}
  <input type="hidden" name="{{ redirect_field_name }}" val\
ue="{{ redirect_field_value }}" />
  {% endif %}

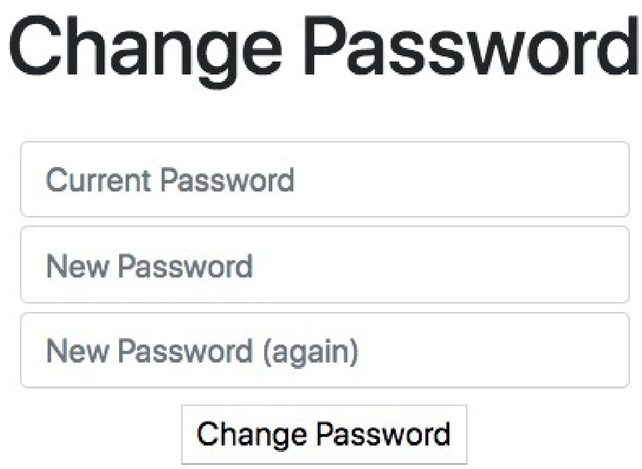
 <button class="btn btn-lg btn-primary btn-block" type="sub\
mit">{% trans "Sign Up" %} &raquo;</button>
</form>
```



Edit *password_change.html* file in the templates *allauth/account/* folder. Replace the *form* element with these lines:

templates/allauth/account/password_change.html

```
<form method="POST" action="{% url 'account_change_password\
' %}" class=" form-account password_change">
        {% csrf_token %}
        {% include 'form_snippet.html' %}
        <button class="mt-1" type="submit" name="action">{% trans \
"Change Password" %}</button>
</form>
```



25.5 Details

25.5.1 Configuration options

The *Allauth* package offers quite a bit configuration options. Let's take a look at what we used:

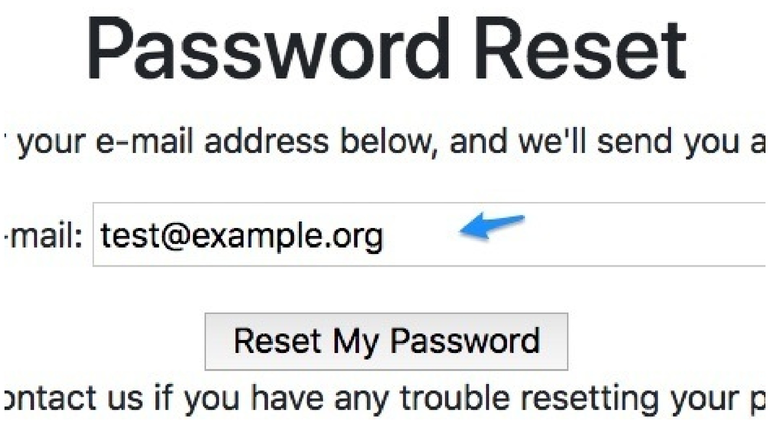
mysite/settings.py

```
SITE_ID = 1 # < here
EMAIL_BACKEND = 'django.core.mail.backends.console.EmailBac\
kend' # < here
LOGIN_REDIRECT_URL = '/' # < here
```

`SITE_ID = 1` has to match the site added in *admin/sites/site/*. In this case we use the default *example.com* site.

With `EMAIL_BACKEND` variable we tell Django to write emails to the standard output instead of trying to send the emails. This is useful for development but for production you should use something like *SendGrid*. We will do that in the *Sending Emails* chapter.

You can try this by visiting *accounts/password/reset/*:



Emails are written in the standard output stream

```
...
Subject: [example.com] Password Reset E-mail
From: webmaster@localhost
To: test@example.org
...
```

With `LOGIN_REDIRECT_URL` we redirect the user to the home page after a successful login. Otherwise you would be redirected to a *profile* page that doesn't exist by default.

Check out the official documentation for more *configuration* options: https://samuli.to/Django-Allauth.

25.5.2 Adding the paths

In the *urls.py* file we included all *django-allauth* paths with one line:

mysite/urls.py

```
from django.contrib import admin
from django.urls import path, include # < here
from myapp import views as myapp_views

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', myapp_views.index, name='index'),
    path('accounts/', include('allauth.urls')), # < here
]
```

Here is a list for all paths it provides:

All django-allauth paths

```
accounts/signup/
accounts/login/
```

```
accounts/logout/
accounts/password/change/
accounts/password/set/
accounts/inactive/
accounts/email/
accounts/confirm-email/
accounts/confirm-email/<key>/
accounts/password/reset/
accounts/password/reset/done/
accounts/password/reset/key/<uidb36>/
accounts/password/reset/key/done/
accounts/social/login/cancelled/
accounts/social/login/error/
accounts/social/signup/
accounts/social/connections/
```

Note that we only customized all *major* templates but you can take a look at the *templates/allauth* folder and go through all of them.

25.5.3 django-widget-tweaks

With *django-widget-tweaks* you can manipulate form field rendering in templates. I use it to add the *form-control* CSS class to input fields:

templates/form_snippet.html

```
{% load widget_tweaks %}

{% for field in form %}
<div class="fieldWrapper mb-1">
  {{ field.errors }}
  {% if field.field.widget.input_type != 'checkbox' %}
    <label class="sr-only" for="{{ form_field.auto_id }}">{\
{ form_field.label }}</label>
    {{ field|add_class:"form-control" }} <!-- here -->
  {% else %}
    {{ field.label_tag }}
    {{ field }}
  {% endif %}
</div>
{% endfor %}
```

I use *if* statement to exclude the `form-control` CSS class from *checkboxes*.

Read more about the *django-widget-tweaks* package: https://samuli.to/Widget-Tweaks

25.6 Summary

- With `django-allauth` package you can add an *account* management functionality without writing any custom views.

- In development environment you can use *EMAIL_BACKEND* variable to write emails to the standard output for easy debugging.
- With `django-widget-tweak` package you can change form field rendering in templates.

26. Authorization

This chapter covers

- How to manage user permissions with *groups*
- How to manage access using *decorators*

26.1 Setup

Terminal

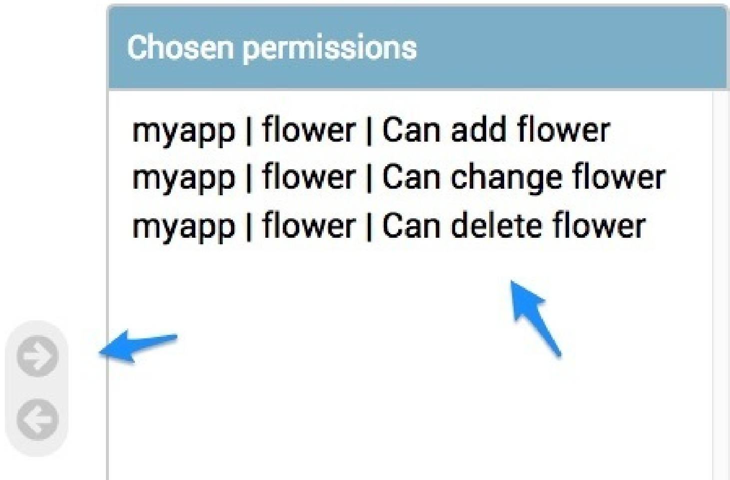
```
cp -fr 24-Object-Manipulation 26-Authorization
cd 26-Authorization
source ../venv/bin/activate
```

26.2 Adding the Editor group

Visit *admin* and add a new “Editor” group using the “+Add” link:

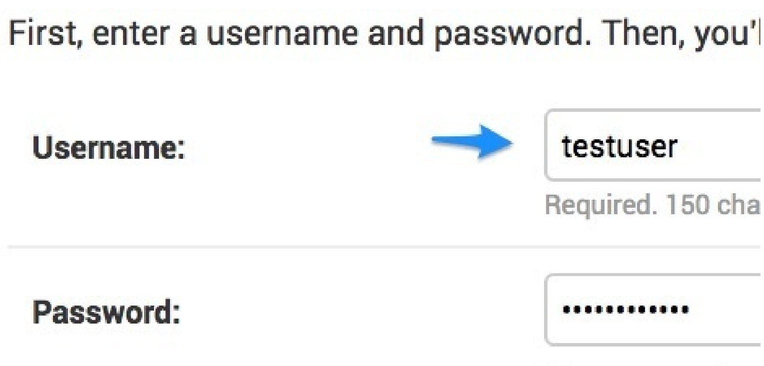


Select the following *permissions* and click *save*:

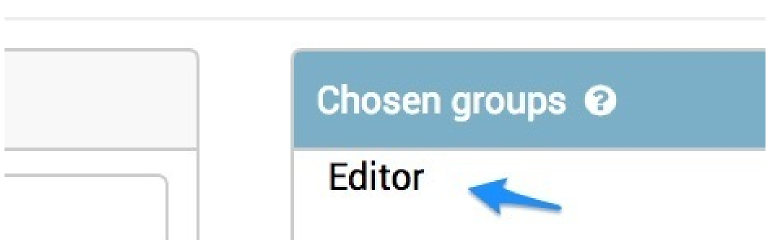


26.3 Creating a test user

Visit *admin* and add a new user using the “+Add” link.



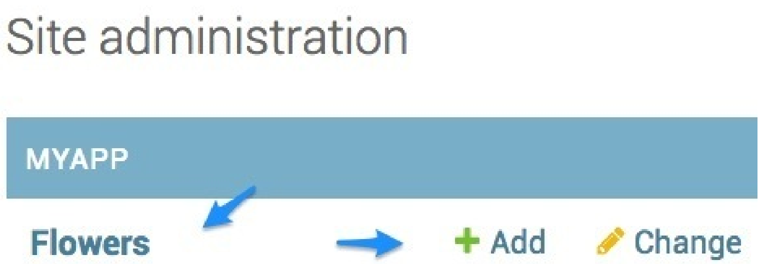
Add user to the *Editor* group:



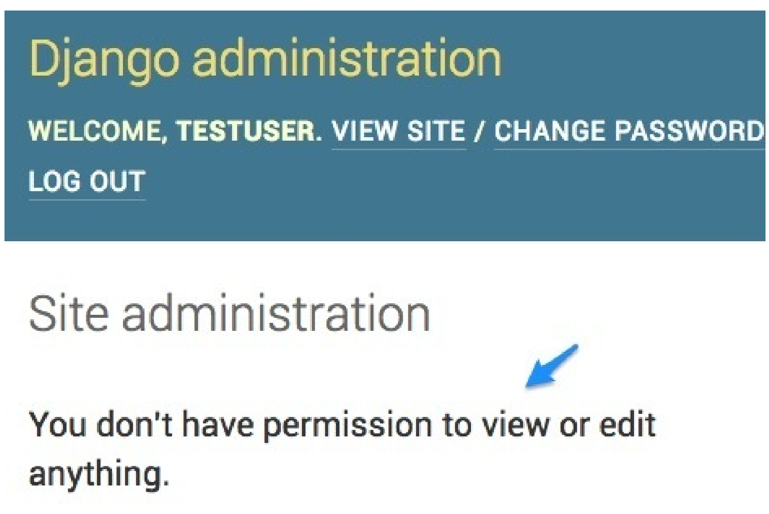
Check *Staff status* checkbox and save:



Open another browser and log in the *testuser* in *admin*. Our *testuser* has now permissions to manage *Flower* items:



If you remove the *testuser* from the *Editor* group, then the admin interface would show the following message:



Our *testuser* can still login to the admin because the *Staff status* is still enabled for the account.

26.4 Using permissions

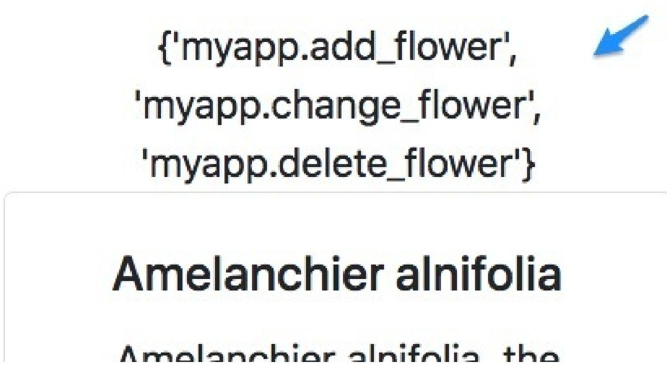
Edit myapp *index.html* page and add if statements to check the user permissions:

myapp/templates/myapp/index.html

```
{{ request.user.get_all_permissions }} <!-- here -->

<div class="card-columns">
{% for flower in flowers %}
...
      {% if perms.myapp.change_flower %} <!-- here -->
        <a href="{% url 'edit' pk=flower.pk %}" class="card\
-link">Edit</a>
      {% endif %}
      {% if perms.myapp.delete_flower %} <!-- here -->
        <a href="{% url 'delete' pk=flower.pk %}" class="ca\
rd-link">Delete</a>
      {% endif %}
...
{% endfor %}
</div>
```

`{{ request.user.get_all_permissions }}` shows the *current* user permissions.



Now only users with correct permissions will see the *Edit* and *Delete* links.

26.5 Using decorators

But currently anyone can manage flowers using our custom forms. Let's restrict access with *decorators*.

Edit myapp *views.py* file and add the decorators:

myapp/views.py

```
from django.contrib.auth.decorators import permission_requi\
red # < here
...
```

```python
def index(request):
        ...
@permission_required('myapp.add_flower') # < here
def create(request):
        ...

@permission_required('myapp.change_flower') # < here
def edit(request, pk=None):
        ...

@permission_required('myapp.change_delete') # < here
def delete(request, pk=None):
        ...
```

Now only accounts with the right permissions can access these views.

26.6 Details

26.6.1 Authentication vs authorization

Authentication is about verifying a user. *Authorization* is about restricting or allowing access to resources.

With *Groups* you can give multiple *permissions* to users at once. The *Editor* group contains permissions for *adding*, *changing* and *deleting* flowers. The user who belongs to the *Editor* group will get all these permissions.

`{{ request.user.get_all_permissions }}` reveals the *machine names* for the current user permissions:

User permissions

```
{'myapp.delete_flower',
'myapp.change_flower',
'myapp.add_flower'}
```

You can use `perms.PERMISSION` in templates to access the current user permissions:

Checking user permissions

```
{% if perms.myapp.change_flower %}
...
{% endif %}
```

26.6.2 Controlling access with decorators

Decorators allow us to dynamically alter a function or a class. Django provides some useful decorators related to user access: https://samuli.to/Auth-Decorators.

Using a decorator

```
@permission_required('myapp.add_flower')
def create(request):
        ...
```

Another useful is the `login_required` decorator:

@login_required decorator

```
@login_required
def profile(request):
        ...
```

In this case you would have to be logged-in to access the profile page. Otherwise the visitor will be redirected to a URL specified with `settings.LOGIN_URL`.

26.7 Summary

- You can *group* permissions and assign users to these groups.
- Current user permissions are available in templates using the `{{ perms }}` template variable.
- `{{ request.user.get_all_permissions }}` displays all permissions for the current logged-in user.
- `@permission_required()` decorator checks if the current user has a particular permission. This is a convenient way to restrict access to specific views.
- `@login_required` is a more general decorator that requires that user has to be *logged-in*.

27. Creating an image gallery

This chapter covers

- How to upload images
- How to serve the images in localhost
- How to show the images in a grid using Bootstrap 4 album

27.1 Setup

Terminal

```
cp -fr 15-Base-Project 27-Image-Gallery
cd 27-Image-Gallery
source ../venv/bin/activate
```

27.2 Installing pillow

Install the *pillow* package:

Terminal

```
pip install pillow
```

27.3 Configuring media variables

Edit mysite app *settings.py* file and specify `MEDIA_URL` and `MEDIA_ROOT` variables:

mysite/settings.py

```
STATIC_URL = 'static'
MEDIA_URL = 'media'
MEDIA_ROOT = 'media/'
```

27.4 Adding ImageField

Edit myapp *models.py* file and add an *ImageField*:

myapp/models.py

```
from django.db import models

class Flower(models.Model):
    title = models.CharField(max_length=255, default='')
    description = models.TextField(default='')
    image = models.ImageField(default='', blank=True, uploa\
```

```
d_to='images') # < here

    def __str__(self):
        return self.title
```

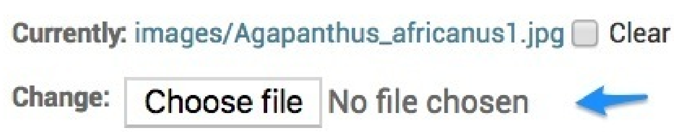
Run migrations:

Terminal

```
python manage.py makemigrations
python manage.py migrate
```

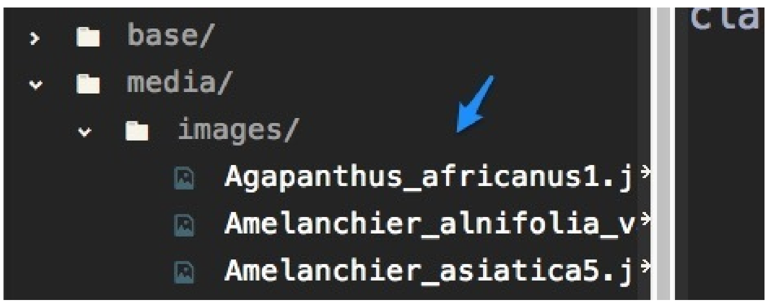
27.5 Adding images to flowers

Visit *admin*, edit the flowers and add some images:



You can find example images in this folder: https://samuli.to/Flowers.

Images are uploaded in the *mediaimages/* folder:



27.6 Using the static helper function

Edit mysite app *urls.py* file and use the `static()` helper function:

mysite/urls.py

```
from django.contrib import admin
from django.urls import path
from myapp import views as myapp_views

from django.conf import settings # < here
from django.conf.urls.static import static # < here
```

```
urlpatterns = [
    path('admin/', admin.site.urls),
    path('', myapp_views.index, name='index'),
] + static(settings.MEDIA_URL, document_root=settings.MEDIA\
_ROOT) # < here
```

27.7 Adding the grid

Edit myapp *index.html* file and replace the contents with these lines:

myapp/templates/myapp/index.html

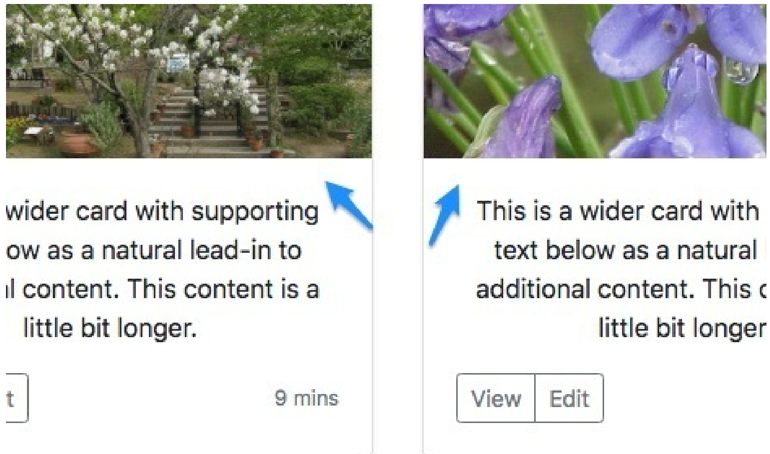
```
{% extends 'base/base.html' %}
{% block content %}

<div class="album py-5">
    <div class="container">
        <div class="row">
            {% for flower in flowers %}
            <div class="col-md-4">
                <div class="card mb-4 shadow-sm">
                    <img class="card-img-top" src="{{ flowe\
r.image.url }}"
                        alt="Card image cap">
                    <div class="card-body">
                        <p class="card-text">This is a wide\
r card with supporting text below as a natural lead-in to
                            additional content. This conten\
t is a little bit longer.</p>
                        <div class="d-flex justify-content-\
between align-items-center">
                            <div class="btn-group">
                                <button type="button" class\
="btn btn-sm btn-outline-secondary">View</button>
                                <button type="button" class\
="btn btn-sm btn-outline-secondary">Edit</button>
                            </div>
                            <small class="text-muted">9 min\
s</small>
                        </div>
                    </div>
                </div>
            </div>
            {% endfor %}
        </div>
    </div>
</div>

{% endblock %}
```

You can find the grid markup in here: https://samuli.to/Grid.

Visit home page and you should see the *album* grid:



27.8 Details

You need to install the *Pillow* library to add an *ImageField*:

myapp/models.py

```
image = models.ImageField(default='', blank=True, upload_to\
='images')
```

`upload_to='images'` stores the uploaded images in the `media/images/` folder.

In the development phase you can serve these user-uploaded files using `static()` helper function:

myapp/urls.py

```
urlpatterns = [
    path('admin/', admin.site.urls),
    path('', myapp_views.index, name='index'),
] + static(settings.MEDIA_URL, document_root=settings.MEDIA\
_ROOT) # < here
```

This function works only in *debug* mode. You have to have `DEBUG = True` configured in the *settings.py* file. With Heroku platform we will serve the media files from an Amazons AWS bucket later in the book.

Use `{{ flower.image.url }}` to access image URLS in templates:

Accessing the image url

```
<img class="card-img-top" src="{{ flower.image.url }}"
```

The grid is just a basic Bootstrap *album*: https://samuli.to/Bootstrap-Album

27.9 Summary

- *Pillow* package adds image uploading and processing capabilities.
- *MEDIA_ROOT* is the physical path to the images.
- *MEDIA_URL* is the URL path you use to access the media files.
- You can use `static()` function to serve the files in debug mode. In production environment you have to implement other ways to serve the images.
- In templates the image URLS are accessed with the familiar dot "." notation: `{{ flower.image.url }}`.

In the next chapter we generate *smaller* images and *crop* them.

28. Adding image thumbnails

This chapter covers

- How to create thumbnails with ImageKit

28.1 Setup

Terminal

```
cp -fr 27-Image-Gallery 28-Image-Thumbnails
cd 28-Image-Thumbnails
source ../venv/bin/activate
```

28.2 Installing ImageKit

Terminal

```
pip install django-imagekit
```

Edit mysite app *settings.py* file and add *imagekit* to the *INSTALLED_APPS* list:

mysite/settings.py

```
INSTALLED_APPS = [
        ...
    'base',
    'myapp',
    'imagekit', # < here
]
```

28.3 Adding the thumbnail field

Edit myapp *models.py* file and add the *image_thumbnail* field:

mysite/models.py

```
from django.db import models
from imagekit.models import ImageSpecField # < here
from pilkit.processors import ResizeToFill # < here

class Flower(models.Model):
    title = models.CharField(max_length=255, default='')
    description = models.TextField(default='')
    image = models.ImageField(default='', blank=True, uploa\
d_to='images')
    image_thumbnail = ImageSpecField(source='image',
        processors=[ResizeToFill(350, 200)],
        format='JPEG',
        options={'quality': 60}) # < here
```

Edit myapp *index.html* file and replace `{{ flower.image.url }}` with `{{ flower.image_thumbnail.url }}`:

myapp/templates/myapp/index.html

```
<img class="card-img-top" src="{{ flower.image_thumbnail.ur\
l }}"
```

Visit the homepage to generate the image thumbnails. They will be served from paths like this:

mediaCACHE/images/images/Agapanthus.jpg

28.4 Details

It's very useful to generate *thumbnails* for images. You can always add links to the original images if needed. We use *ImageKit* to *crop* and *resize* the images. The thumbnails are generated as the page where the images are used is accessed the first time.

Using the original uploaded images can result in very heavy pages. For example *Amelanchier_asiatica5.jpg* that I used for testing was *4.3MB*. Image processing reduced that size to *18.2KB*!

`ImageSpecField` is similar to `ImageField` but it automatically applies the image processing we specify:

ImageSpecField does the image proccessing

```
image = models.ImageField(default='', blank=True, upload_to\
='images')
image_thumbnail = ImageSpecField(source='image',
        processors=[ResizeToFill(350, 200)],
        format='JPEG',
        options={'quality': 60})
```

`source='image'` is the *original* image field. We can add different *processors* (https://samuli.to/Processors) to manipulate the image. *ResizeToFill* resizes and crops the image. Here we also specify image *format* and *compression*.

You can access the thumbnail URL using the dot "." notation in templates: `{{ flower.image_thumbnail.url }}`.

28.5 Summary

- Creating *thumbnails* can reduce the image sizes substantially.
- *ImageKit* package enables a selection of image processing tools.

29. Deploying on Heroku

This chapter covers

- How to deploy to Heroku

29.1 Setup

Create a folder outside the *projects* folder:

Terminal

```
mkdir deployments
cd deployments
mkdir heroku
cd heroku
python3 -m venv venv
source venv/bin/activate
pip install django django-heroku gunicorn
pip freeze > requirements.txt
django-admin startproject mysite .
python manage.py runserver
```

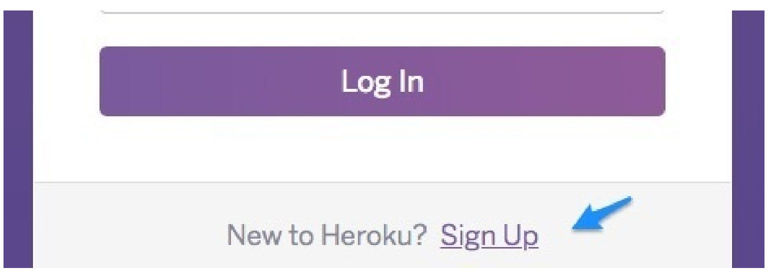
django-heroku package installs some dependencies like *psycopg2* for PostgreSQL support and *whitenoise* for serving static files straight from the app.

Terminal

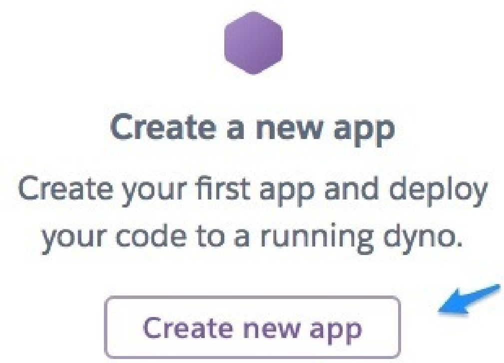
```
├── deployments # < here
│   ├── heroku # < here
├── projects
```

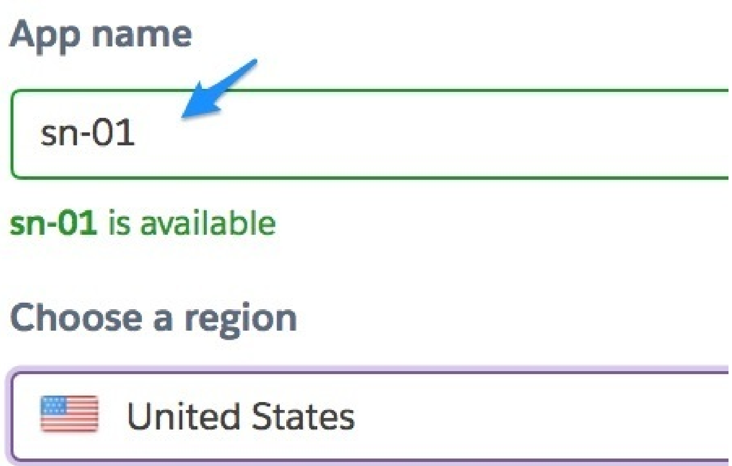
29.2 Creating a Heroku app

Visit https://samuli.to/Heroku and create an account:



Press *Create new app*:





Rest of the chapter shows *sn-01* as the app *name*. Replace it with the name of your app.

29.3 Installing Heroku CLI

29.3.1 Installation in Windows

Visit https://samuli.to/Heroku-CLI and download the Windows installer.

29.3.2 Installation in macOS

Terminal

Terminal

```
xcode-select --install
brew install heroku/brew/heroku
```

29.3.3 Installation in Ubuntu

Terminal

```
sudo snap install --classic heroku
```

29.3.4 Authenticating with a browser

Use `heroku login` in terminal to login:

Terminal

```
heroku login
heroku: Press any key to open up the browser to login or q \
to exit:
Logging in... done
Logged in as user@example.org
```

29.4 Creating a Procfile

Create a file called *Procfile* in the project root and write this line in it:

Procfile contents

```
web: gunicorn mysite.wsgi
```

29.5 Updating the settings.py file

Edit *settings.py* file and import *django_heroku* package on the top and change *DEBUG* and *ALLOWED_HOSTS* variables:

mysite/settings.py

```
import django_heroku # < here
import os

DEBUG = False # < here
ALLOWED_HOSTS = ['sn-01.herokuapp.com'] # < here
```

Add the following lines at the bottom of the file:

mysite/settings.py

```
django_heroku.settings(locals())

try:
    from .local_settings import *
except ImportError:
    pass
```

Create a *local_settings.py* file:

mysite/local_settings.py

```
DEBUG = True
ALLOWED_HOSTS = []
```

29.6 Creating the repository

Visit https://samuli.to/Git and install Git.

Create a *.gitignore* file in the site root:

.gitignore file

```
venv
local_settings.py

db.sqlite3
*.pyc
__pycache__/
*.py[cod]
.DS_Store
```

Visit https://samuli.to/Dj-Gitignore too see more comprehensive *.gitignore* example.

Initialise git repository and push it:

Terminal

```
git init
git add .
git commit -m "Initial"
heroku git:remote -a sn-01
git push heroku master
```

Run *migrate* and create a *superuser*:

Terminal

```
heroku run python manage.py migrate
heroku run python manage.py createsuperuser
```

Visit your app admin pages in https://sn-01.herokuapp.com/admin/.

Note: we don’t see the welcome screen on the frontpage because the production site is *not* in debug mode. You get “*The requested URL / was not found on this server.*” instead because we don’t have a *view* for the homepage.

29.7 Pushing changes

Let's add a homepage and some CSS styling. The *django-heroku* package installs the *Whitenoise* package that allows your web app to serve its own static files. Check out the next chapter on how to serve *static* files and *user-uploaded* files from *Amazon AWS*.

Terminal

```
django-admin startapp blog
```

Add an *index* view:

blog/views.py

```
from django.shortcuts import render

def index(request): # < here
    return render(request, 'blog/index.html')
```

Create an *index.html* file with this content:

blog/templates*blog*index.html

```
{% load static %}
<!DOCTYPE html>
<html lang="en">
<head>
    <title>Blog</title>
    <link rel="stylesheet" href="{% static 'blog/css/site.c\
ss' %}">
</head>
<body>
    <div id="content">
        <h1>Home</h1>
    </div>
</body>
</html>
```

You have to create the folder structure: *blogtemplatesblog*.

Create a *site.css* file with this content:

blog/static*blog*css/site.css

```
h1 { color: red;}
```

You have to create the folder structure: *blogstaticblogcss/*.

Edit *urls.py* file and add the *index* path:

mysite/urls.py

```
from django.contrib import admin
from django.urls import path

from blog import views # < here
```

```
urlpatterns = [
    path('admin/', admin.site.urls),
    path('', views.index, name='index') # < here
]
```

Add ‘blog’ to the *INSTALLED_APPS* list:

mysite/settings.py

```
INSTALLED_APPS = [
        ...
    'django.contrib.staticfiles',
    'blog', # < here
]
```

Terminal

```
git add .
git commit -m "Add Blog app"
git push heroku master
```

Visit the production site homepage and you should see this:



Note: we didn’t have to run “*heroku run python manage.py migrate*” because we didn’t make any changes that require database updates.

29.8 Updating the database

Let’s create a Post model and update the database:

blog/models.py

```
from django.db import models

class Post(models.Model):
    title = models.CharField(max_length=255, default='')
```

Register it in admin.py:

blog/admin.py

```
from django.contrib import admin
```

```
from .models import Post

admin.site.register(Post)
```

Run *local* migrations:

Terminal

```
python manage.py makemigrations
python manage.py migrate
python manage.py createsuperuser
python manage.py runserver
```

Login and create a post item to see that it works locally before you push it.

Push the changes:

Terminal

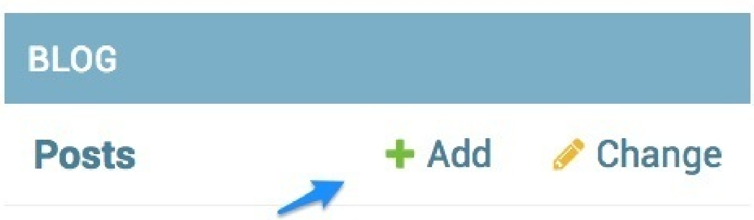
```
git add .
git commit -m "Add Post model"
git push heroku master
```

Apply changes to the remote database:

Terminal

```
heroku run python manage.py migrate
```

Visit your heroku app admin page and add content:



29.9 Summary

- *django-heroku* adds settings configuration. This includes things like *DATABASE_URL* so that you don't have to add database configuration manually. It also install some extra packages like *whitenoise* that allows

you to serve static files directly from the app without using *Nginx*, *Amazon S3* or any other similar solution.

- Use “*pip freeze > requirements.txt*” to generate a dependency list. These will be installed automatically when you push the code.
- Remember to set `DEBUG = False` and configure `ALLOWED_HOSTS` variable in the *settings.py* file for production environments.
- It’s useful to create multiple settings files like *local_settings.py* to add environment specific configuration.
- *Heroku CLI* allows you to interact with the platform using a command line. It requires *GIT* to work.
- You can run remote commands with “*heroku run <command>*”. For example, if you make changes to the database schema, you should run “*heroku run python manage.py migrate*”.
- Use “*git push heroku master*” to push changes to the platform. Check out the “Heroku Pipelines” chapter on how to create a proper deployment flow.

30. Using Amazon AWS to serve files

This chapter covers

- How to serve *static* assets and *user-uploaded* files from an Amazon bucket

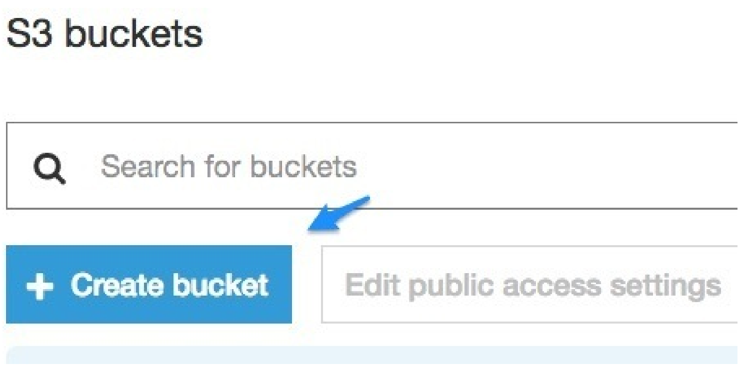
30.1 Setup

Use the project from the "Heroku Deployment" chapter to test this.

30.2 Creating an Amazon AWS bucket

Visit https://samuli.to/AWS and create an account.

Visit https://samuli.to/S3 and add a *bucket*:

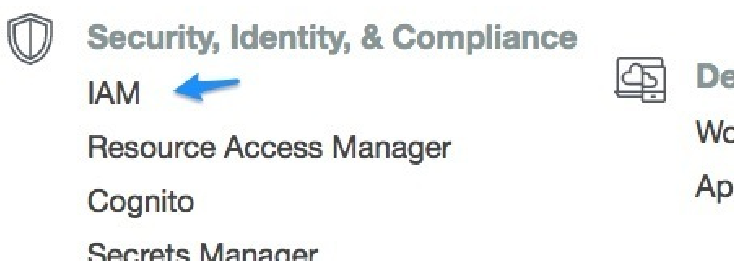




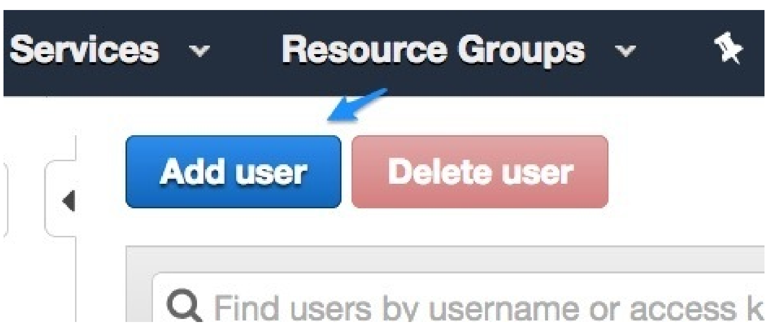
Click *Next* for the rest of the settings and hit *Create bucket*.

30.3 Setting up permissions

Visit *Services* and click *IAM* under the *Security, Identity & Compliance* label:



Click *Users* and *Add user*:

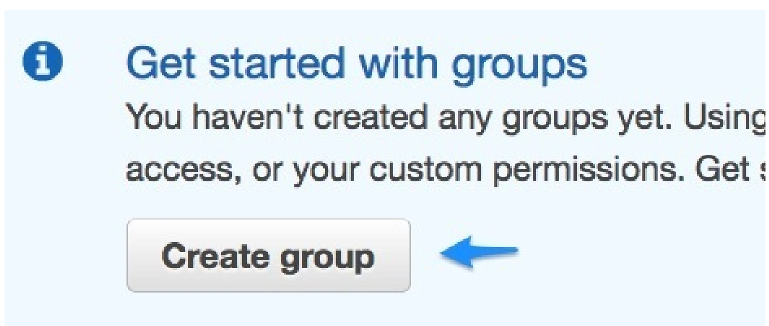




Check *Programmatic access*:

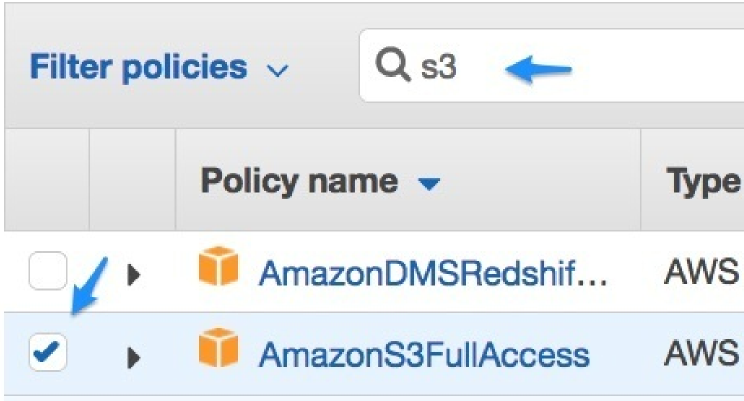


Create a new *group*:





Check *AmazonS3FullAccess*:



Click *Next: Tags*:



Click *Next: Review*:



Click *Create user*:



We will use this information in the *settings.py* file:

Access key ID	Secret access
AKIAI5OPW7S44Y6UGKXA	yzJd9y3aV03Z KSE Hide

30.4 Updating settings.py file

Update *settings.py* file and add the configuration:

mysite/settings.py

```
django_heroku.settings(locals())

AWS_ACCESS_KEY_ID = 'ACCESS_KEY'
AWS_SECRET_ACCESS_KEY = 'SECRET'
AWS_STORAGE_BUCKET_NAME = 'sn-test-01'

AWS_DEFAULT_ACL = None

AWS_LOCATION = 'static'
AWS_MEDIA_LOCATION = 'media'

STATIC_URL = 'https://%s.s3.amazonaws.com/%s/' % (AWS_STORA\
GE_BUCKET_NAME, AWS_LOCATION)

STATICFILES_STORAGE = 'storages.backends.s3boto3.S3Boto3Sto\
rage'
DEFAULT_FILE_STORAGE = 'mysite.storages.MediaStorage'

try:
    from .local_settings import *
except ImportError:
    pass
```

Create a *storages.py* file and fill it with these lines:

mysite/storages.py

```
from django.conf import settings
from storages.backends.s3boto3 import S3Boto3Storage

class MediaStorage(S3Boto3Storage):
    location = settings.AWS_MEDIA_LOCATION
    file_overwrite = False
```

30.5 Adding an image field to the Post model

Edit blog app *models.py* file and add an *ImageField*:

blog/models.py

```
from django.db import models

class Post(models.Model):
    title = models.CharField(max_length=255, default='')
    image = models.ImageField(default='', blank=True, uploa\
d_to='images') # < here
```

30.6 Installing packages

Install *packages* and push:

Terminal

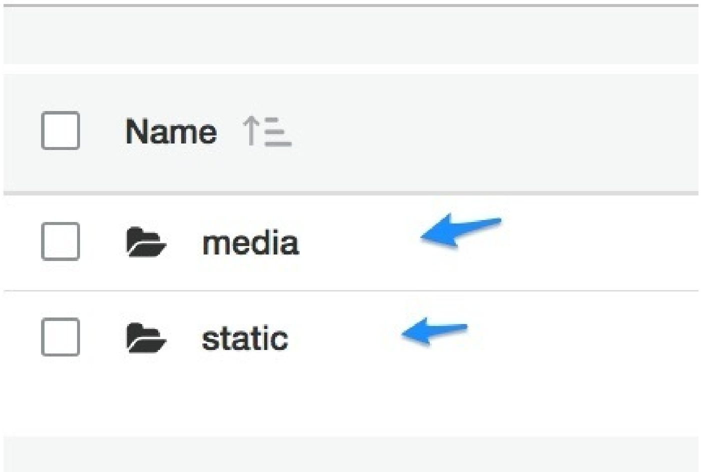
```
pip install django-storages boto3 pillow
python manage.py makemigrations
python manage.py migrate
pip freeze > requirements.txt
git add .
git commit -m "Add django-storages, boto3, pillow and Post \
model image field"
git push heroku master
heroku run python manage.py migrate
```

Visit the production site in https://YOUR_APP.herokuapp.com/ and create a Post with an image.

The post image will be now served from an URL like this: *sn-test-01.amazonaws.com/media/images/Agapanthus.png*

Open the page source code and you will see that the static files are now served from URLS like this: *sn-test-01.s3.amazonaws.com/static/admin/css/base.css*

In the bucket folder you now have separate folders for *media* and *static* files:



30.7 Summary

- *Boto3* is an Amazon software development kit that allows Python programs to use services like Amazon S3.
- It's not uncommon to serve *static assets* and *user-uploaded files* from external sources.
- Amazon S3 can also be integrated with a *content delivery network* like Amazon CloudFront https://samuli.to/Amazon-CloudFront.

31. Setting up Heroku pipelines

This chapter covers

- How to create a continuous deployment workflow with Heroku pipelines

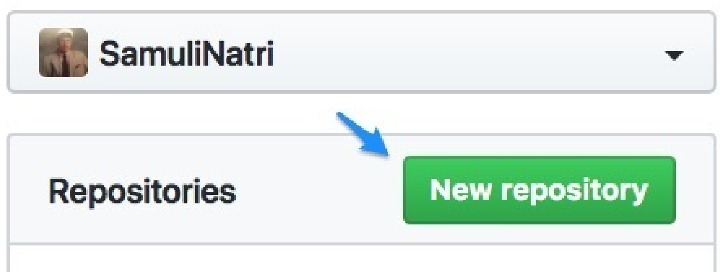
31.1 Setup

Use the project from the “Heroku Deployment” chapter to test this.

31.2 Creating a GitHub repository

Visit https://samuli.to/GitHub and create an account.

Create a new repository:



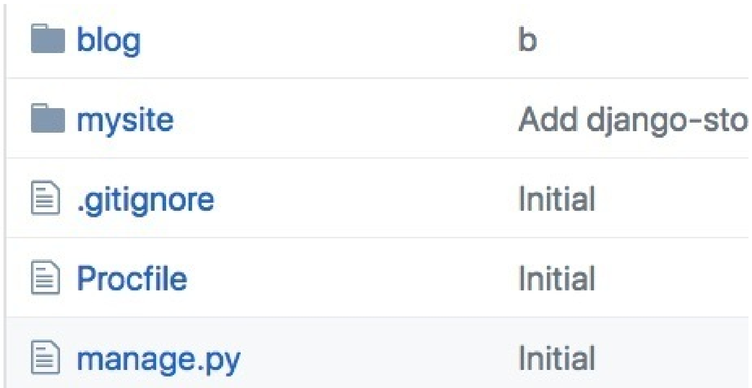


Go to your project folder. Add a *remote* and *push* the code to GitHub:

Terminal

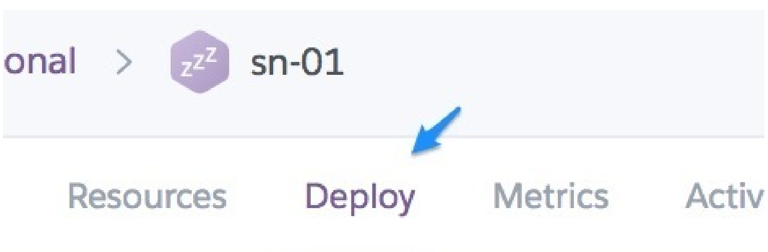
```
git remote add origin git@github.com:SamuliNatri/sn-01.git
git push -u origin master
```

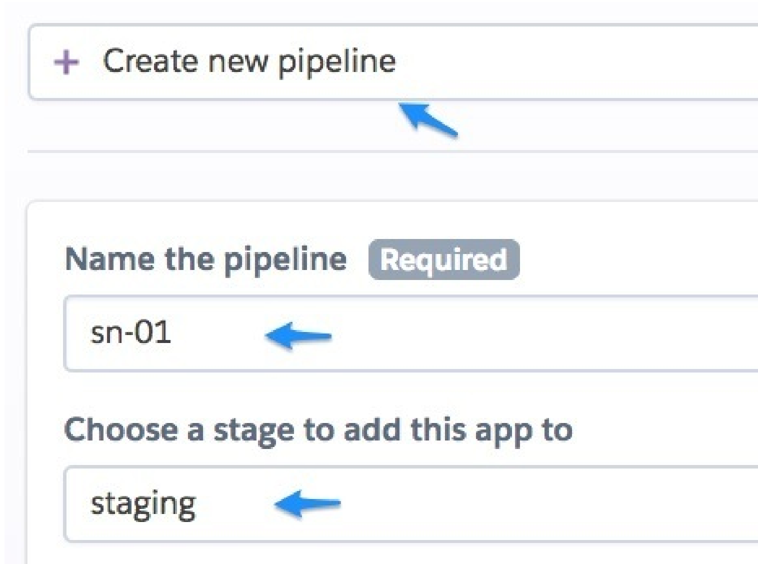
Refresh the GitHub page and you should see the project code:



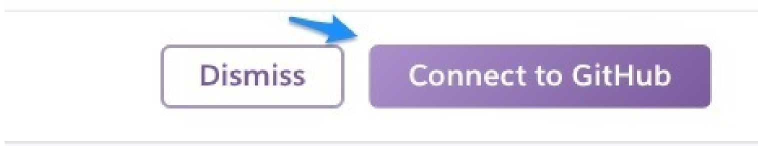
31.3 Creating a pipeline

Visit your Heroku app *Deploy* page and create a *pipeline*:

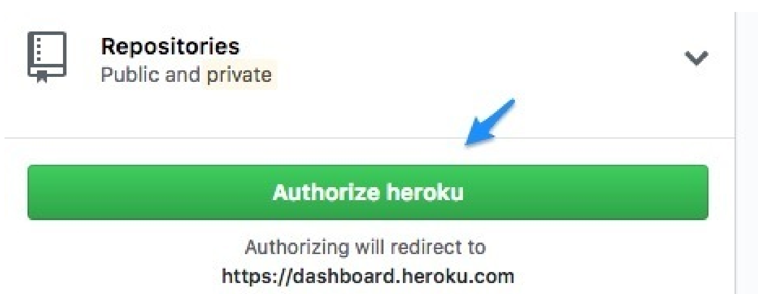




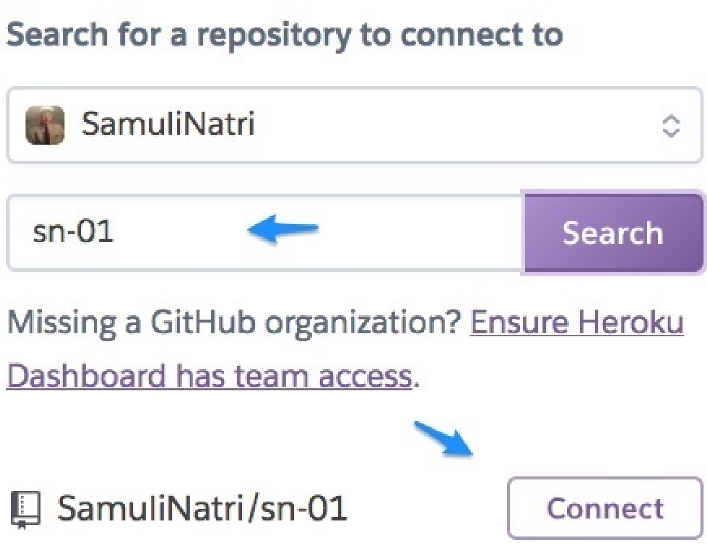
Press *Connect to GitHub*:



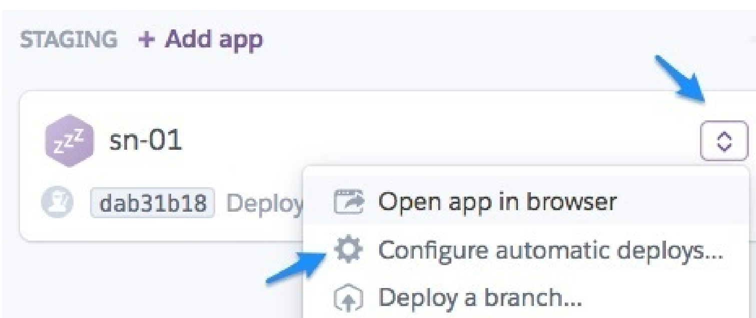
Login to GitHub and *Authorize heroku*:

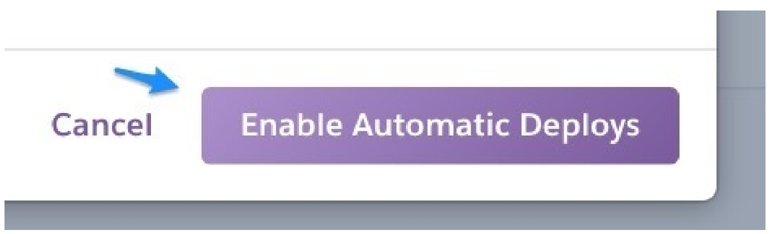


Search for the repository and *Connect* it:



Visit the *Pipeline* page and *Enable Automatic Deploys*:





31.4 Testing deployment

Edit the *index.html* template and change the “Home” text:

blog/templates/blog/index.html

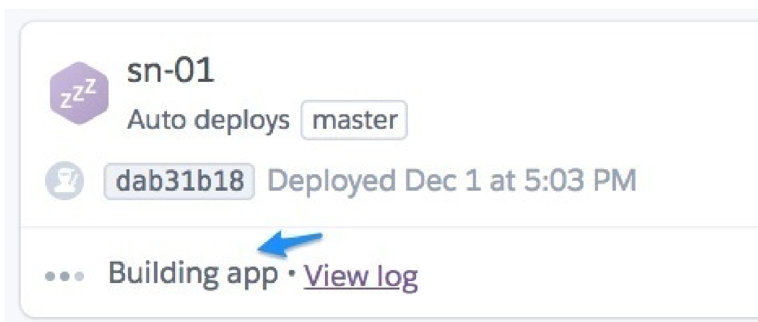
```
{% load static %}
<!DOCTYPE html>
```

```
<html lang="en">
<head>
    <title>Blog</title>
    <link rel="stylesheet" href="{% static 'blog/css/site.c\
ss' %}">
</head>
<body>
    <div id="content">
        <h1>Home (Update)</h1> # < here
    </div>
</body>
</html>
```

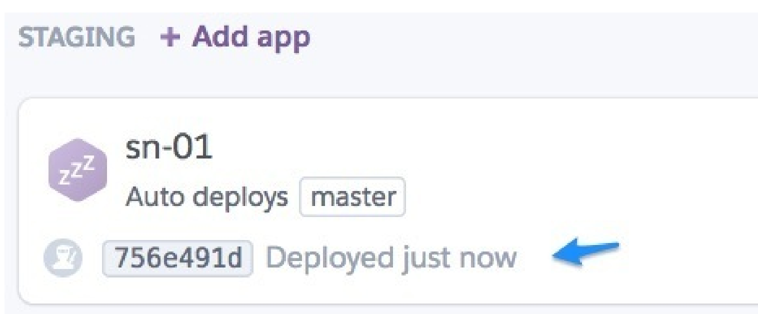
Terminal

```
git add .
git commit -m "Update homepage"
git push
```

In a moment you will see “Building app” text on the page:



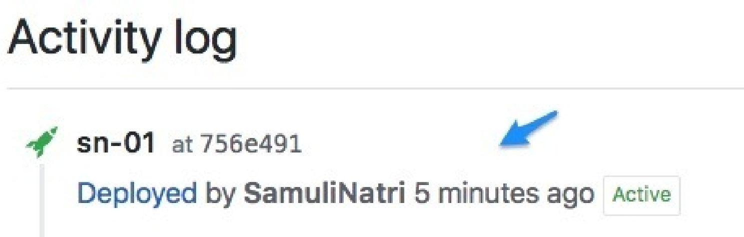
And “Deployed..” text when the deployment is ready:



Visit the app URL and you should see the changes:

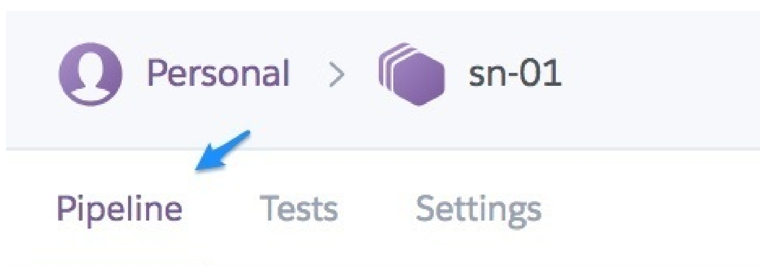


These deployments will also show in the GitHub *Deployments* section:

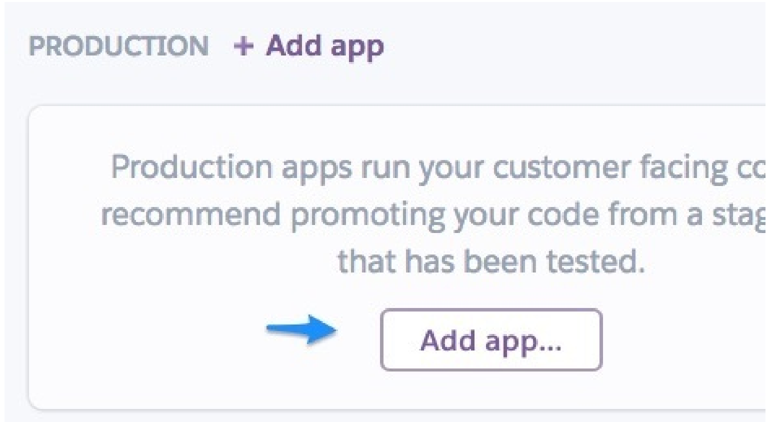


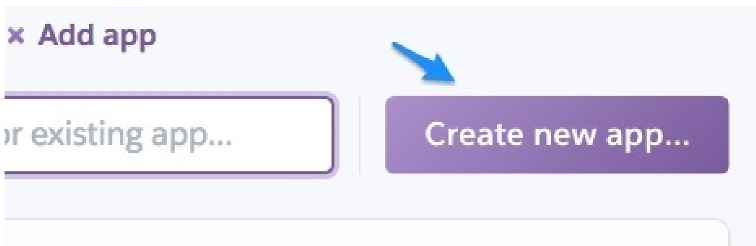
31.5 Adding a production app

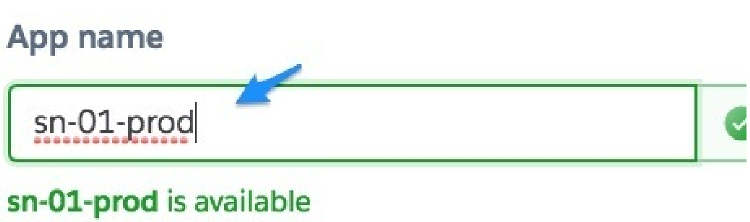
Visit the *Pipeline* page:



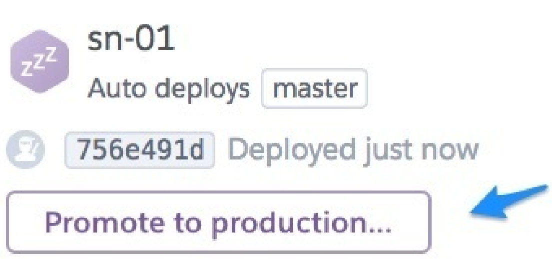
Add a *Production* app:







Press your staging app *Promote to production* button:

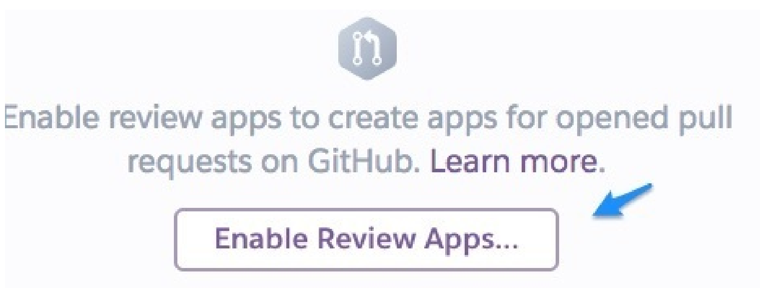


Visit your production app homepage and it should look like the staging app homepage:

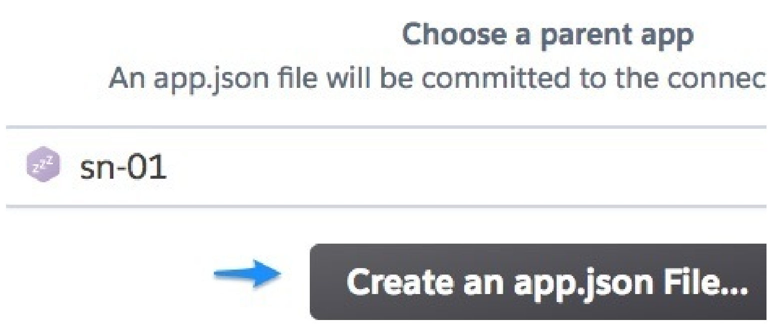


31.6 Enabling review apps

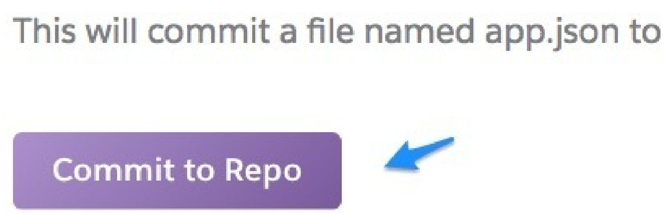
Visit the *Pipeline* page and press *Enable Review Apps*:



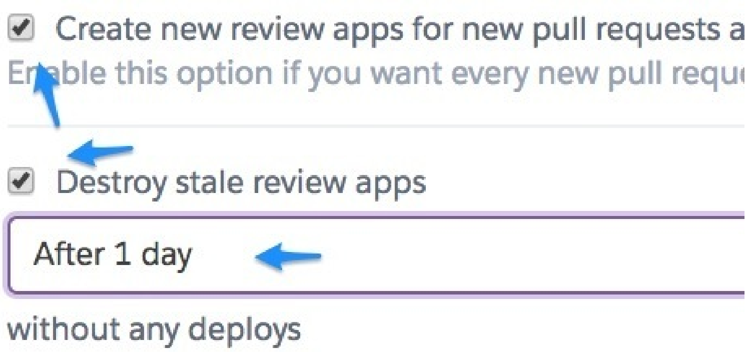
Create an *app.json* file:



Scroll to the bottom and press *Commit to Repo*:



Check *Create new review apps…automatically* and *Destroy stale review apps*. Press *Enable*:



Note that review apps may incur dyno and add-on *charges*: https://samuli.to/Review-Apps!

You can also *not* check the *Create new review apps…automatically* option and create preview apps manually on the *Pipeline* page.

31.7 Using pull requests

Let's make a change and create a *pull request*.

Pull changes and create a branch:

Terminal

```
git pull
git checkout -b new_homepage
```

We need to pull the *app.json* file that the platform added to the repo.

Edit the *index.html* template and make some changes:

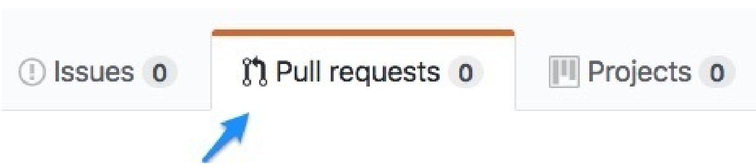
blog/templates/blog/index.html

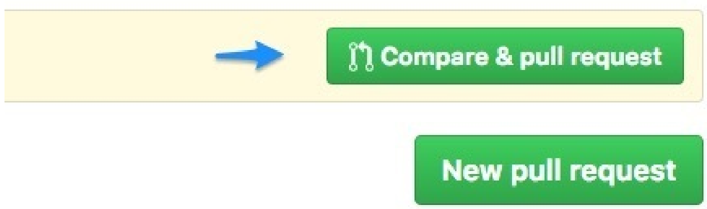
```
<div id="content">
    <h1>NEW FANCY HOMEPAGE</h1> <!-- here -->
</div>
```

Terminal

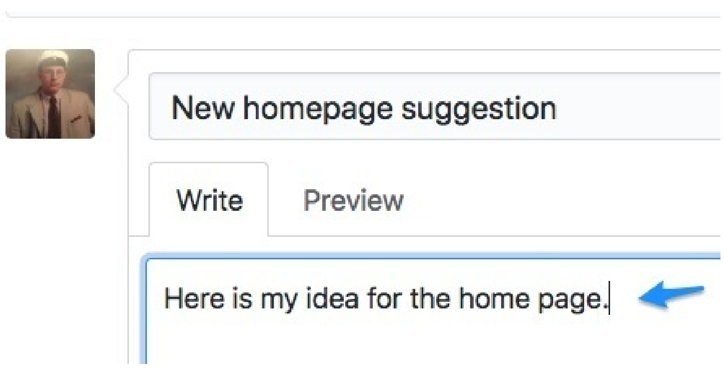
```
git add .
git commit -m "New homepage suggestion"
git push --set-upstream origin new_homepage
```

Use link in the *Terminal* to create a *Pull request* or visit the *Pull requests* page on GitHub:

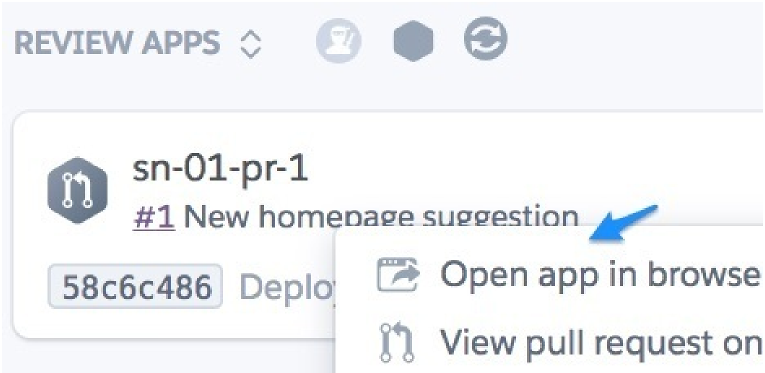




Write a description and create a *Pull request*:



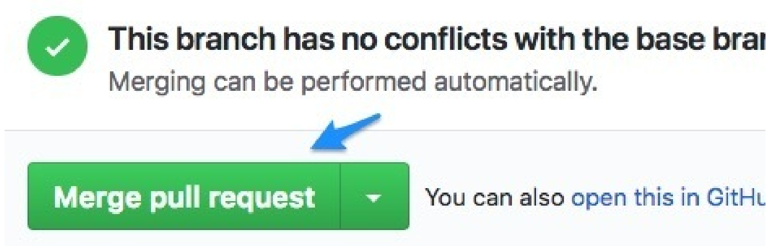
Visit the *Pipeline* page and click *Open app in browser* after the preview app is ready:



You can now evaluate the pull request in the preview app:

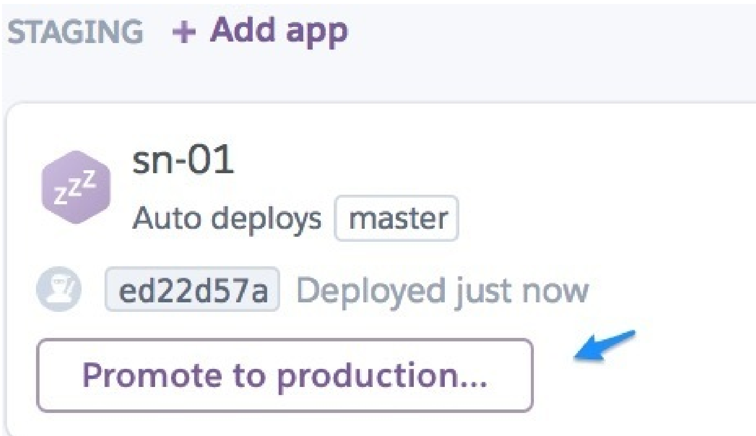


Visit *GitHub* and merge the pull request:

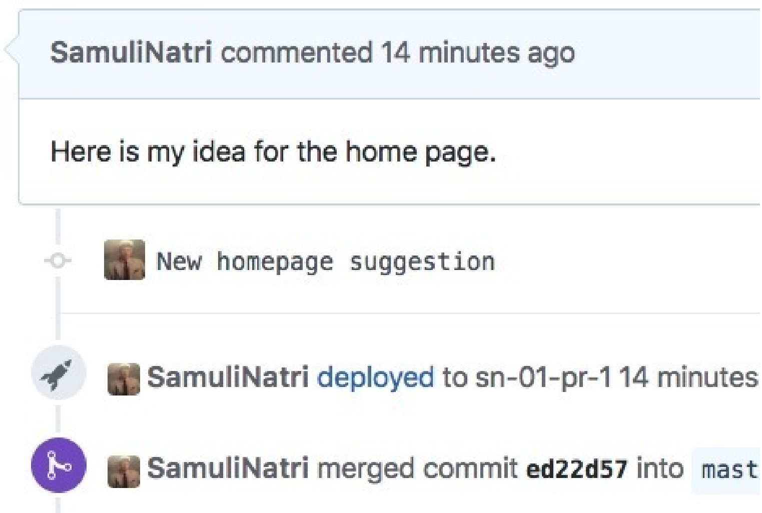




Visit the *Pipeline* page and wait for the staging app to be deployed. Press *Promote to production* and the new fancy home page is now live:



The *pull request* and *merging* flow is also visible in GitHub:



31.8 Deleting the branch

We don't need the *new_homepage* branch anymore since it's now merged to the *master* branch:

Terminal

```
git branch
git checkout master
git pull
git branch -d new_homepage
```

31.9 Summary

- Heroku provides a nice *continuous delivery workflow* out of the box.
- *Review apps* allow you to test GitHub pull requests with disposable Heroku apps.

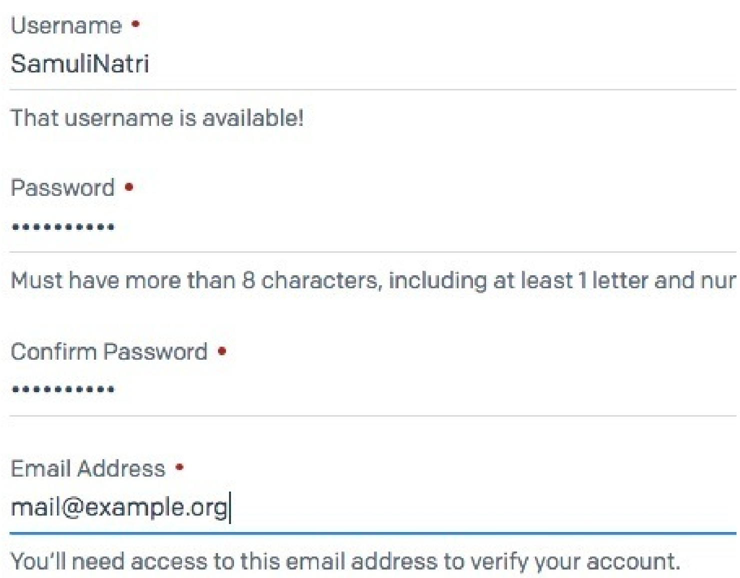
32. Sending emails with SendGrid

This chapter covers

- How to send emails with SendGrid

32.1 Creating an account

Visit https://samuli.to/SendGrid and create an account:



Copy the base project:

Terminal

```
cp -fr 15-Base-Project 32-Sending-Emails
cd 32-Sending-Emails
source ../venv/bin/activate
```

Edit *settings.py* file and add the following configuration using the *username* and *password* you provided in the sign-in process:

mysite/settings.py

```
EMAIL_HOST = 'smtp.sendgrid.net'
EMAIL_HOST_USER = 'sendgrid_username'
EMAIL_HOST_PASSWORD = 'sendgrid_password'
EMAIL_PORT = 587
EMAIL_USE_TLS = True
```

Test the mail in the *interactive interpreter*:

Interactive interpreter

```
python manage.py shell
>>> from django.core.mail import send_mail
>>> send_mail('Subject here', 'Here is the message.', 'admi\
n-mail@gmail.com', ['some-other-mail@gmail.com'], fail_sile\
ntly=False)
```

You should now receive the email in your inbox:



32.2 Summary

- Sending emails with *SendGrid* is just matter of creating an account with the service and adding the right configuration to the settings.py file.

Licenses